UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE INGENIERÍA TELEMÁTICA

PROYECTO FIN DE CARRERA

Trazabilidad de mercancías en movilidad

Laura Carmona Acebedo

Tutor: Vicente Luque Centeno

Madrid, 2010

Agradecimientos

La finalización del proyecto final de carrera representa la transición entre dos etapas, la diferencia entre tener el título de ingeniero y no tenerlo. Este momento, tan deseado desde que se empiezan los estudios universitarios, llega tras una ilusión inicial que se transforma en años de esfuerzo y sacrificio, motivo por el que es un orgullo haberlo alcanzado finalmente. Orgullo que se ve incrementado por haber sido la Escuela Politécnica Superior de la Universidad Carlos III de Madrid el lugar en el que he realizado estos estudios.

La realización de este proyecto no sólo no solo supone para mí el superar la etapa universitaria, sino que ha cambiado muchos otros aspectos de mi vida, motivo por el que sentirme doblemente feliz.

Quiero agradecer de forma muy especial todo el apoyo que me han dado mis padres desde que comenzó la etapa universitaria para mí, ya que sin ese apoyo que me han dado siempre, en todos los aspectos de la vida, no hubiera llegado hasta aquí. Gracias por apoyarme y aconsejarme en los malos momentos, que no han sido pocos como bien sabéis, así como compartir conmigo los buenos, que por otra parte compensan los malos con creces. Solo vosotros entendéis lo importante que es este momento. GRACIAS.

Gracias también a toda la gente, familiares y amigos, que me ha apoyado, ayudado y animado siempre durante esta etapa que ahora finaliza. Mención especial merecen también todos los profesores que en algún momento han contribuido a mi formación, y especialmente a Vicente Luque Centeno por su orientación y consejos en el desarrollo de este proyecto.

Muchas gracias a todos.

Resumen

Durante las últimas décadas, hemos sido testigos de los importantes avances logrados en el campo de la tecnología celular. En la actualidad, no sólo los teléfonos móviles son usados para hacer llamadas o enviar mensajes de texto, sino que también integran diferentes funcionalidades, como el receptor GPS, navegación web, reproductor de música, fotografía digital, etc. Este tipo de teléfonos son llamados *smartphones*.

Desde que estos dispositivos se hicieron asequibles para la mayoría de la gente, su uso se ha extendido mucho. No sólo los usuarios finales poseen este tipo de dispositivos, sino que también las empresas los están empezando a usar como herramientas de trabajo.

El objetivo de esta herramienta es ayudar a gestionar y controlar las tareas de los procesos de negocio para una compañía de transporte de mercancías. Por un lado, los transportistas usan *smartphones* con sistema operativo Android para realizar sus tareas asignadas. Tienen acceso a sus expediciones (entregas, recogidas, mensajes) de sus dispositivos de forma que pueden consultar y reportar información asociada a sus actividades diarias. Esto no sólo ahorrará mucho tiempo, sino que también hace más fácil y rápido el proceso: no es necesario para los transportistas ir a la central para recoger y devolver sus hojas de ruta. Las actividades realizadas por cada uno de ellos se registran automáticamente en la base de datos del sistema. Por otra parte, el personal de la planta puede realizar un seguimiento, casi de manera instantánea, de las entregas y recogidas mediante el módulo web de la aplicación.

Android proporciona una fácil integración con diferentes componentes. Usando la herramienta de TTM, los conductores pueden iniciar otras das aplicaciones que se han integrado en ella, las cuales pueden ser muy útiles para los mismos. Estas aplicaciones son *Places Directory* y *Gasolineras España*.

Abstract

During the last decades, we have been witnesses of the important advances achieved in the cell technology field. Currently, mobile phones not only are used for making calls or send SMSs, but also they integrate lots of diferent functionalities, such as GPS receiver, web browsing, music player, digital photography, etc. These sort of phones are called *smartphones*.

Since these devices became affordable for most of the people, its use has spread a lot. Not only final users own this sort of devices, but also companys are including these devices as work tools.

The purpouse of this tool is to help in manage and control tasks of bussiness processes for a shipping company. On one hand, truck drivers use *smartphones* with OS Android to carry out their assigned tasks. They have access to their shipments (deliveries, collections, messages) from their devices in order to read and report information associated to their daily activities. This not only save much time, but also makes easier and faster the process: it is not necessary for the drivers neither to go to the plant to pick up nor to return their roadmaps. The activities done for each of them are registered automatically in the system database, too. On the other hand, staff at the plant can track, almost in an instant way, deliveries and collections using the web component of this application.

Android supplies an easy integration of different components together. Using TTM tool, drivers can launch another two applications that have been integrated in it, which can be very usefull for them. These applications are *Places Directory* and *Gasolineras España*.

Índice general

Índice de figuras

Índice de códigos fuente

Índice de cuadros

Capítulo 1

INTRODUCCIÓN

El 3 de abril de 1973, en la Sexta Avenida frente al hotel Hilton de Nueva York, el ingeniero de Motorola Martin Cooper llamó con un teléfono móvil a su contrincante Joel S. Engel, de Bell Labs, en la que se considera la primera llamada desde un móvil de la historia. Martin Cooper, considerado por muchos como el padre de la tecnología móvil, actualmente utiliza un Motorola Droid con sistema operativo Android [1].

La tecnología celular ha experimentado un gran avance desde el lanzamiento al mercado del primer teléfono móvil en 1983 hasta los terminales de los que disponemos hoy en día. Las distintas necesidades así como los avances logrados dieron lugar a generaciones tecnológicas bien diferenciadas cuyas capacidades marcaron a su vez la evolución tanto del hardware como del software de los terminales. Dichas generaciones resumidas brevemente son las siguientes [2]:

Generación 1G

- La transferencia analógica y estrictamente para voz son características identificadoras de esta generación. La calidad de los enlaces de voz era muy baja y la velocidad de conexión no era mayor a 2400 bauds. Basadas en FDMA, había una limitación en el número de usuarios que podían usar el servicio simultáneamente. Además la seguridad era inexistente.

Generación 2G

- La mejora más importante que aportó esta tecnología es que las transferencias son digitales. Usó a su vez TDMA para permitir que hasta ocho usuarios utilizaran los canales. Los protocolos empleados en los sistemas 2G soportan velocidades de

información más altas para voz, pero limitados en comunicaciones de datos. Se pueden ofrecer otros servicios tales como datos, fax y SMS. Las tecnologías predominantes son: GSM, IS-136, CDMA y PDC.

Generación 2.5G

- La entrada del GPRS en escena se considera la evolución del 2G al 3G. El GPRS puede usarse para servicios WAP, SMS, MMS y para servicios de comunicación por Internet como el email y el acceso a la web. Con este nuevo sistema los canales pasaron a compartirse por más de un usuario lo que permitía comunicaciones más eficientes y baratas.

Generación 3G

- Se conoce a esta generación como un conjunto de estándares de telecomunicaciones. Los servicios 3G permiten el uso de voz y datos simultáneamente a altas velocidades, lo que hace esta generación apta para aplicaciones multimedia y altas transmisiones de datos. La tecnología usada en estas redes es UMTS [3] [4].

Una vez conocida la evolución de la tecnología celular y las posibilidades que ofrecía en cada momento, podemos entender mejor la evolución sufrida por los terminales móviles que fueron adaptándose de forma que aprovecharan dichas posibilidades.

A pesar de que la tecnología celular fue concebida estrictamente para voz, los teléfonos móviles han ido incorporando cada vez más funcionalidades. Algunas de estas funcionalidades son: reproductores de música y vídeo, fotografía digital, acceso a correo electrónico, agenda, navegación web, videojuegos, etc. Estos terminales son conocidos como *smartphones*.

Aunque el objetivo ha sido siempre crear dispositivos más potentes y funcionales con un tamaño cada vez más reducido, esta tendencia en cuanto al tamaño de los terminales parece haber cambiado en los últimos años. La razón la podemos encontrar en las nuevas capacidades que ofrecían las redes y servicios 3G. Comenzaron a aparecer dispositivos con teclado QWERTY y pantallas con mayor resolución, especialmente desde la salida al mercado de *smartphones* de pantalla táctil con teclado virtual. De esta forma, actividades como leer el correo o navegar por la web desde el terminal móvil resultan más cómodas y sencillas para el usuario.

Los teléfonos móviles ya se habían convertido en una herramienta imprescindible tanto a nivel personal como empresarial, pero gracias a los avances anteriormente comentados se ha extendido más su uso, especialmente en el ámbito empresarial. Mientras que hace unos años eran utilizados sobre todo por perfiles con necesidad de comunicación por voz como por ejemplo comerciales, personal de gestión, etc., actualmente se emplean también en otros campos como la seguridad, logística, etc. En general, toda actividad que suponga un "trabajo de campo" no sólo es susceptible de ser movilizada sino que supone un aumento de productividad para la empresa que la lleva a cabo y por tanto de sus beneficios.

Este aumento en el uso de los *smartphones* como herramientas de trabajo es debido a que las empresas desarrolladoras de los sistemas operativos que incorporan los terminales ponen a disposición de terceros unas APIs y Frameworks con los que implementar aplicaciones para dichos dispositivos. De esta forma, los usuarios disponen de una gran variedad de aplicaciones, así como la posibilidad de obtenerlas a medida. Otro de los factores que han impulsado la creación de software por terceros es la incorporación de brújula, receptor de GPS y acelerómetro a los terminales, de forma que se pueden ofrecer servicios basados en localización geográfica, lo cual es muy atractivo para algunos sectores.

Google es la compañía que desarrolló el sistema operativo Android, el cual fue presentado oficialmente a finales del año 2007. No ha sido, sin embargo, hasta el año 2009 que ha comenzado a sufrir un considerable impulso en el mercado gracias a los distintos dispositivos puestos en venta que lo incorporaban. Los primeros fabricantes que decidieron incluir Android en sus terminales se han visto también beneficiados de esta decisión. Dos ejemplos de ello son HTC o Motorola, esta última pasando de tener unas pérdidas de 231 millones de dólares en el primer trimestre del 2009 a obtener 69 millones de dólares en beneficios en el primer trimestre de este año [5] [6].

A continuación se muestran unas gráficas en las que podemos ver la evolución del crecimiento de Android. Los porcentajes en ambas gráficas muestran el uso por sistema operativo referido únicamente al grupo de los *smartphones*. El primer gráfico corresponde con los datos recogidos hace un año, cuando el porcentaje de *smartphones* con sistema Android era de tan sólo un 3 %. En el segundo gráfico podemos apreciar como Android ha ido aumentando su crecimiento durante los últimos 6 meses hasta alcanzar un 25 % de uso.

Se espera que la popularización de Android aumente aún más en este año 2010 debido a su incorporación no solo en *smartphones* sino en otro tipo de dispositivos

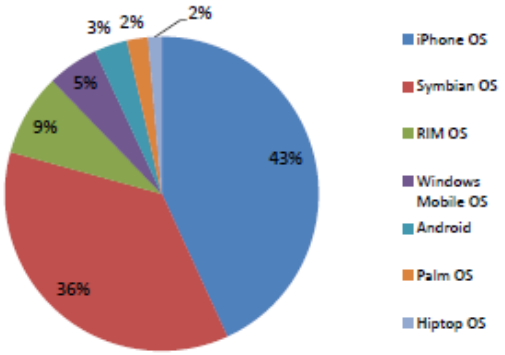

Figura 1.1: Porcentaje de uso por sistema operativo en Abril 2009 [7].

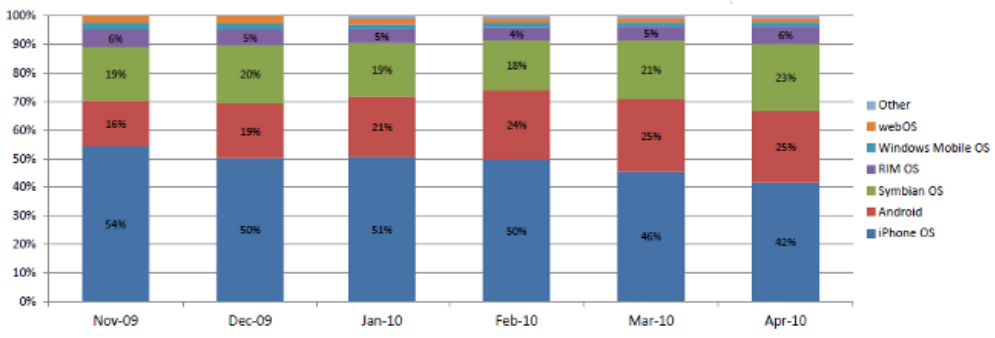

Figura 1.2: Porcentaje de uso por sistema operativo Noviembre 2009 - Abril 2010 [8].

móviles como los netbooks, e-readers, tablets, televisiones, etc. Según Jeff Huber, vicepresidente senior de ingeniería de Google [9], en el momento en que se ha escrito este documento 12 OEMs distintas comercializan 34 dispositivos que portan Android y 60.000 terminales con Android son vendidos y activados cada día. Cabe también destacar el Android Market como factor de ayuda a la popularización de la plataforma. Este servicio permite la distribución de aplicaciones de terceros tanto de carácter gratuito como de pago, de forma que cualquier usuario puede descargar a su terminal gran cantidad de aplicaciones que complementen las ya incluidas en el software original del dispositivo. J. Huber hizo también pública la cifra de 38.000 aplicaciones disponibles para descarga en el Android Market, sin tener en cuenta *ringtones* y *wallpapers* [10].

Android se lanza bajo la licencia Apache [11], lo cual significa que puede ser usado, copiado, estudiado, cambiado y redistribuido libremente, y por tanto el código fuente es accesible por cualquier desarrollador. Google proporciona además un SDK y toda una serie de herramientas con las que poder implementar aplicaciones para Android en Java, lenguaje de programación empleado por este sistema operativo.

Por todo esto, se ha elegido Android para llevar a cabo la movilización de un sistema de forma que se incremente la productividad en la realización de actividades que lo componen. La aplicación que se ha desarrollado en este PFC como ejemplo está enfocada al sector del transporte de mercancías. Se pretende que los transportistas tengan acceso a sus expediciones (entregas, recogidas y notificaciones) desde terminales móviles para poder consultar y reportar información asociada a su actividad diaria. Ésta quedaría registrada de forma automática en el sistema de forma que el personal encargado en la central podría realizar un seguimiento casi en directo de los repartos/recogidas.

Además, aprovechando una característica de Android que es la reutilización y reemplazo de componentes, se pueden integrar varias soluciones distintas basadas en localización geográfica que complementen y faciliten la labor de los transportistas, como por ejemplo aplicaciones de búsqueda de gasolineras, talleres, estado del tráfico, etc.

1.1. Justificación del proyecto

Funcionalidades como las que aquí se pretenden desarrollar se empezaron a introducir en España hace unos años, sin embargo, su implantación en las empresas de transporte aún no está generalizada. La trazabilidad de las mercancías continúa siendo una tarea pendiente. Muchas empresas de transportes asignan las expediciones a los vehículos de la flota siguiendo un proceso manual que continúa, al cabo de la jornada,

por la revisión de los albaranes de entrega y la introducción de los datos en un sistema informático para su procesamiento por lotes. Por otra parte, el entorno notablemente competitivo del sector exige la necesidad de conocer en tiempo real el estado de las expediciones según estas se van produciendo, de cara a poder tratar proactivamente posibles incidencias.

Las principales ventajas que se pueden obtener mediante la implantación de un sistema de estas características en dichas empresas son: la reducción de costes y la mejora de la calidad del servicio. La reducción de costes viene dada principalmente por la mejora de la gestión de la flota y la actividad de los transportistas. En muchos casos las empresas no hacen un uso adecuado de su flota y esto les lleva a tener vehículos infrautilizados. Al tener constancia de la localización se consigue un mejor aprovechamiento de los mismos, pudiendo aumentar su rentabilidad evitando las vueltas en vacío por ejemplo, así como reducir el consumo del combustible. La actividad de los repartidores se ve reducida en tiempo, lo cual supone igualmente un aumento de los beneficios. Por otra parte, el hecho de mantener un control de las mercancías de forma individual estando en comunicación y localización contínua supone una mejora importante y perceptible en la calidad del servicio.

Para negocios en los que la rapidez es clave, poder reaccionar ante imprevistos de cualquier tipo de manera inmediata es vital para aumentar la satisfacción y la confianza de la clientela, cuya exigencia es cada vez mayor y demandan una gestión óptima en los envíos. Con este tipo se soluciones se tiene la mayor garantía de poder conseguirlo. Además, sería posible informar al destinatario de posibles imprevistos notificando, por ejemplo, que el pedido puede llegar tarde y facilitando una hora de entrega aproximada ya que no sólo es importante entregar a tiempo sino que, cuando esto no es posible debido a una incidencia o imprevisto, es muy importante avisar cuanto antes, y permitir que quien espera recibirlo pueda tomar las medidas apropiadas.

Las empresas del sector han intentado proveerse de este tipo de tecnologías que les aporten un aumento en sus beneficios. Algunas de ellas, como por ejemplo Correos, Nacex o Fedex, ofrecen un seguimiento de las expediciones, pero la lectura de dichas mercancías se realiza únicamente en puntos muy concretos que suelen coincidir con los almacenes de origen, transitarios y de destino. Por otra parte, también es posible encontrar sistemas en los que se realiza reporte del estado de la mercanía en tránsito. Sin embargo, éstos requieren de la instalación de aparatosos equipos en los vehículos llamados *Data Communication Terminals*, desde donde se envían dichos datos.

En las soluciones existentes se pueden encontrar algunas que realizan funciones

similares a las de esta herramienta, pero empleando otro *hardware* y *software*, es decir, son aplicaciones cerradas, corriendo sobre PDAs en los mejores casos, con pocas posibilidades de crecimiento y mejora. Por otra parte, es posible encontrar aplicaciones Android que realizan funciones de seguimiento de rutas, de reporte o de visualización de mapas, pero en ningún caso se encuentran todas las funcionalidades agrupadas en una misma herramienta. Incluso alguna de dichas herramientas ni siquiera está pensada para el sector del transporte, sino para otros ámbitos como por ejemplo el deporte.

En cuanto a seguimiento de rutas por ejemplo, encontramos el famoso *MyTracks* del propio Google. Esta aplicación permite grabar rutas realizadas y visualizarlas posteriormente junto con ciertas estadísticas en la sección "My Maps". Sin embargo, es el propio usuario el que debe comenzar y finalizar la grabación de la ruta, guardarla y enviarla, lo cual es automático en la herramienta aquí desarrollada. Con "MyTracks" podría cubrirse la funcionalidad de visualización de rutas, pero por ejemplo se perdería la función añadida de ver en ese mismo mapa el estado de las expediciones. Además, sería necesaria una cuenta de usuario distinta para cada transportista a la hora ver los mapas, mientras que con la herramienta *TMM* un usuario administrador puede ver los de cada transportista simplemente seleccionándolos desde una tabla. Otra aplicación similar sería "InstaMapper" [43], la cual se instala en el dispositivo móvil y envía coordenadas a un servidor, de forma que posteriormente se pueden ver las rutas en una interfaz web. Los incovenientes vuelven a quedar a la vista: por cada usuario se necesita una cuenta y el resto de funcionalidades de el sistema aquí desarrollado desaparece.

La ventaja de este sistema, en resumen, es que aunaría en una sola herramienta todas las funcionalidades definidas en la página 56. Además, el hecho de desarrollarse sobre la plataforma Android haría a la aplicación muy escalable, no limitando la funcionalidad a la comunicación entre el dispositivo y el servidor, sino que el usuario dispondría de una herramienta muy potente gracias a la posibilidad de integrar fácilmente otros módulos como por ejemplo información sobre meteorología, talleres, tráfico, etc. Este es un punto que hay que tener en cuenta de cara al futuro, ya que Android es una plataforma que está creciendo a gran velocidad gracias al aporte tanto de aplicaciones de terceros como por parte del propio Google.

1.2. Objetivos

El objetivo de este PFC es lograr el diseño y desarrollo de una aplicación compuesta por varios módulos intercomunicados entre sí que permitan agilizar y mejorar la actividad de gestión y seguimiento de expediciones en una empresa de transporte de mercancías.

Los módulos de los que se compondrá la aplicación son los siguientes:

1- Módulo cliente Android:

En cuanto a los objetivos funcionales, el aplicativo debe ser capaz de comunicarse vía web services con un servidor con el que intercambiar los datos de las expediciones, los cuales deben mostrarse en la interfaz de usuario. Además, tendrá un carácter multi-idioma, pudiendo seleccionarse el idioma español o el inglés. El dispositivo enviará también su posición geográfica al servidor cada cierto tiempo para que éste pueda reproducir la ruta seguida por el transportista. Por último, se reutilizará e integrará algún otro módulo o componente que pueda serle útil al usuario en el desarrollo de su actividad.

Teniendo en cuenta la usabilidad, el UI debe ser atractivo, intuitivo y sencillo. Hay que tener en cuenta que el usuario será una persona que no tiene por qué estar necesariamente en posesión de conocimientos avanzados en tecnologías móviles. El esquema de navegación entre pantallas será simple con objetivo de realizar las operaciones en el menor tiempo posible.

La aplicación se desarrollará teniendo presente las pautas marcadas por Google en la documentación sobre Android en cuanto a rendimiento y capacidad de respuesta, de forma que se haga un buen uso de los recursos del sistema.

2- Perfil de gestión web para jefe de flota:

Funcionalmente, este módulo web debe permitir la gestión de usuarios (alta, baja y modificación) en el sistema, así como la consulta de la actividad diaria de cada uno de ellos. Las notificaciones también serán gestionadas a través de este módulo, permitiéndose la creación, modificación y borrado de cada una de ellas. Podrán consultarse los datos referentes a las distintas entregas y recogidas pudiendo filtrarse dichos datos gracias a determinados campos.

Gracias al API de Google Maps, se añaden mapas que muestran al gestor a tiempo casi real las rutas seguidas por cada uno de los transportistas, así como el estado y datos básicos de cada una de sus entregas/recogidas.

La interfaz será, al igual que en el módulo anterior, atractiva y sencilla.

3- Interfaz web para introducción de expediciones en el sistema:

Este tipo de soluciones en las que se movilizan procesos de una empresa suelen ser soluciones de integración, es decir, se lleva a un escenario móvil una parte del sistema de dicha empresa o se genera de cero toda esa parte perteneciente al entorno móvil pero en ambos casos se debe enlazar con el sistema que ya poseía la compañia, en el cual ya se gestionaban los datos. La funcionalidad de este módulo y en algunos casos la del módulo 5 recaería dentro de ese sistema ya existente, pudiendo volverse innecesarios a la hora de integrar la solución con el mismo. Es decir, una empresa de transporte de mercancías ya poseería su propio sistema con el que han manejado siempre la información y a través del cual se recogen los datos de las expediciones. Aún así, se ha implementado este módulo para proporcionar una solución más completa en este PFC.

4- Web services:

Se implementará un módulo de servicios web con el objetivo de enviar a los terminales móviles los datos procedentes de la base de datos, así como recoger y almacenar en la base de datos la información recibida por dichos terminales.

5- Base de datos:

En la base de datos se almacenarán todas las expediciones de los transportistas, así como sus localizaciones geográficas enviadas desde sus dispositivos. Como se ha comentado en el módulo 3, este módulo podría ser prescindible en la implantación de este sistema en un entorno real.

1.3. Contenido de la memoria

El presente documento está estructurado en varios bloques que se describen a continuación:

El primer capítulo contiene una introducción en la que se pretende definir en qué punto tecnológico se encuentran los dispositivos móviles y su integración en la

sociedad. Seguidamente, se expone una motivación o justificación sobre la realización de este proyecto, en la que se explican los antecedentes, las ventajas de la solución que se quiere desarrollar y una comparativa con las herramientas ya existentes. Por otra parte, se incluye también una breve descripción funcional de la aplicación y los objetivos a alcanzar.

En el segundo capítulo se describe la plataforma Android. Evidentemente, no se ha podido cubrir en este documento cada uno de los aspectos de la plataforma al mínimo detalle, pero los apartados de este capítulo han sido seleccionados y desarrollados con la pretensión de ofrecer una mínima base teórica sobre arquitectura, funcionamiento y peculiaridades del sistema. Es decir, cualquier desarrollador sin conocimientos sobre Android necesitaría aprender cada uno de los apartados aquí tratados para crear una aplicación, aunque después necesitara profundizar más en algunos de ellos. Son estos conocimientos los empleados en el desarrollo de la aplicación cliente móvil de este proyecto.

El resto de herramientas usadas en el desarrollo de este proyecto se tratan en este tercer capítulo. Se hablará principalmente sobre el servidor, la base de datos y la especificación SOAP.

En el cuarto capítulo se hace una descripción en profundidad de la herramienta desarrollada en este proyecto, exponiendo tanto los aspectos de análisis y diseño como los de implementación de la funcionalidad mediante el código. En cuanto al análisis y diseño, se explica la arquitectura que sigue el sistema, los casos de uso y el modelo de datos. En la parte del desarrollo, se explican por separado los módulos de los servicios web, cliente Android y las dos aplicaciones web.

Las conclusiones obtenidas como resultado del desarrollo del proyecto se encuentran en el quinto capítulo de esta memoria. En él se exponen tanto las conclusiones finales como las críticas y dificultades. También se incluye una sección sobre posibles líneas a seguir como continuación de este proyecto.

El cálculo de los costes se presenta en el último capítulo.

Seguidamente al desarrollo de los capítulos principales de la memoria se pueden encontrar un glosario de términos usados en la misma, así como los manuales de usuario de la aplicación desarrollada.

Cerrando la presente memoria, aparecen citadas todas las referencias que se han usado como documentación adicional para la creación de esta memoria.

Capítulo 2

PLATAFORMA ANDROID

Esta plataforma fue desarrollada por Google junto con la Open Handset Alliance [12], un consorcio de empresas de tecnología móvil (software y hardware), operadoras de telefonía y distribuidores tan relevantes como Telefónica, T-Mobile, Vodafone, Samsung, HTC, LG, Motorola, Qualcomm, etc. El objetivo de esta alianza es promocionar estándares abiertos para dispositivos móviles, de forma que se incentive su desarrollo y mejore la experiencia del usuario.

Android es una solución completa de software de código libre para teléfonos y dispositivos móviles, es decir, toda aplicación puede hacer uso de cualquiera de las funcionalidades básicas del teléfono, tales como realizar llamadas, enviar sms, usar la cámara, etc. Dado que el código fuente de Android es abierto, puede ser libremente ampliado para incorporar nuevas tecnologías que vayan surgiendo. Es un paquete que engloba un sistema operativo, un entorno de ejecución basado en Java, un conjunto de librerías de bajo y medio nivel y un conjunto inicial de aplicaciones destinadas al usuario final. Se distribuye bajo una licencia Apache versión 2, una licencia libre permisiva que permite la integración con soluciones de código propietario [13]. Está desarrollado sobre un kernel Linux y utiliza una máquina virtual que fue especialmente diseñada para optimizar la memoria y los recursos de hardware en entornos móviles.

Android fue diseñado desde el principio para permitir a los desarrolladores crear atractivas aplicaciones que aprovechen al máximo el potencial de los dispositivos móviles gracias a un completo conjunto de APIs. Google proporciona además un SDK y toda una serie de herramientas con las que poder implementar aplicaciones para Android mediante el lenguaje de programación Java. Se ha hecho especial énfasis en que las aplicaciones creadas por terceros no tengan ningún tipo de desventaja en cuanto a funcionalidad y acceso al dispositivo que las aplicaciones “nativas”que se distribuyen

originalmente con Android [13]. Con dispositivos basados en esta plataforma los usuarios pueden personalizar totalmente el teléfono según sus intereses.

2.1. Características

Las principales características de esta plataforma son:

- Consta de un framework de aplicaciones que permite reutilizar y reemplazar los distintos componentes.
- Máquina virtual Dalvik, la cual está optimizada para dispositivos móviles.
- Navegador integrado basado en el motor de código abierto WebKit.
- Gráficos optimizados gracias a una biblioteca de gráficos 2D y gráficos 3D basados en OpenGL.
- Sistema de gestión de base de datos relacionales (SQLite) para almacenamiento de datos estructurados.
- Soporte para medios con formatos comunes de audio, vídeo e imágenes (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)
- Telefonía GSM (dependiente del hardware)
- Bluetooth, EDGE, 3G, y WiFi (dependiente del hardware)
- Cámara, GPS, brújula, y acelerómetro (dependiente del hardware)
- Entorno de desarrollo muy completo, incluyendo un emulador de dispositivo, herramientas para depurar, perfiles de memoria y rendimiento, y un complemento para el IDE Eclipse.
- Pantalla táctil.
- Android Market permite que los desarrolladores pongan sus aplicaciones, gratuitas o de pago, en el mercado a través de esta aplicación accesible desde la mayoría de los teléfonos con Android.

2.2. Arquitectura

Android está formado por varios componentes, los cuales se describen a continuación comenzando desde la capa más interna:

1. Núcleo Linux

Los servicios base del sistema tales como la seguridad, gestión de memoria, gestión de procesos, stack de red y drivers están basados en Linux. Android usa el núcleo como una capa de abstracción entre el hardware y el resto del software.

2. Entorno de ejecución

El "runtime" de Android se compone de un conjunto de librerías que proporcionan casi todas las funciones de las librerías base de Java y de la máquina virtual Dalvik.

Cada aplicación corre su propio proceso, con su propia instancia de la máquina virtual Dalvik. Ésta está optimizada para requerir poca memoria y está diseñada para permitir ejecutar varias instancias de la máquina virtual simultáneamente, delegando en el sistema operativo subyacente el soporte de aislamiento de procesos, gestión de memoria e hilos. Dalvik no es una máquina virtual Java exactamente puesto que el bytecode con el que opera no es Java bytecode. La herramienta "dx" incluida en el SDK de Android permite transformar los archivos con formato .class de Java compilados por un compilador Java al formato de archivos .dex. La creación de una VM propia es un movimiento estratégico que permite a Google evitar conflictos con Sun por la licencia de la máquina virtual, así como asegurarse el poder innovar y modificar ésta sin tener que batallar dentro del JCP.

Durante el proceso de creación de este documento, Google ha desarrollado la versión 2.2 Froyo de Android y una de las características más destacadas de ella son las mejoras logradas en la VM Dalvik por su creador, Dan Bornstein, y su equipo de ingenieros. Aunque en cada nueva versión se ha intentado mejorar la DVM, realmente no se ha podido hablar de nueva tecnología en la misma hasta ahora. A pesar de que esta capa es invisible a los usuarios finales, la nueva DVM hace que el software ejecute visiblemente más rápido a la vez que hace un menor uso de la batería, hecho también apreciable por los usuarios. Para ello, se ha añadido un compilador en tiempo real (JIT, Just In Time) a la DVM, el cual toma el código, lo analiza y lo traduce de forma más rápida mientras la aplicación continua ejecutándose. De esta forma se ha conseguido que esta versión sea entre 2 y 5 veces más rápida que la anterior 2.1 Eclair. Además, la memoria usada por el JIT de Dalvik es muy poca: el código del propio JIT son menos de 100k y cada proceso que ejecuta en el JIT usa también unos 100k aproximadamente [14] [15].

3. Bibliotecas

Al mismo nivel que el entorno de ejecución, Android incluye un conjunto de bibliotecas C/C++ las cuales son usadas por varios de los componentes del sistema. Éstas son accesibles para los desarrolladores a través del framework de aplicaciones de Android. Estas bibliotecas son:

- libc, una adaptación de la librería C estándar para dispositivos embebidos basados en Linux.
- Biblioteca multimedia, basada en el OpenCORE de PacketVideo [16] que soporta los siguientes formatos: MPEG4, H.264, MP3, AAC, AMR, JPG, and PNG.
- SQLite, un motor de base de datos relacional que está disponible para todas las aplicaciones.
- WebKit como motor de navegación web. Éste es usado como base también por otros navegadores como Safari, Google Chrome o el navegador de S60 de Nokia.
- OPEN GL ES para creación de gráficos 3D.
- Surface Manager, controla el acceso al display y se encarga de formar las capas de gráficos 2D y 3D de múltiples aplicaciones.
- SGL, motor de gráficos 2D.
- FreeType, un motor de fuentes que permite transformar imágenes vectoriales de fuentes en mapas de bits.

4. Framework de aplicaciones

Como ya se ha comentado anteriormente, en Android se ha hecho especial énfasis en que cualquier aplicación desarrollada por terceros tenga las mismas ventajas que las aplicaciones nativas propias del sistema. Así, los desarrolladores tienen acceso completo a los mismos APIs del framework usados por las aplicaciones base. La arquitectura está diseñada para simplificar la reutilización de componentes; cualquier aplicación puede publicar sus capacidades y cualquier otra aplicación puede luego hacer uso de esas capacidades (sujeto a reglas de seguridad del framework). Este mismo mecanismo permite que los componentes sean reemplazados por el usuario. Toda aplicación, por tanto, puede hacer uso de un conjunto de servicios y sistemas, los cuales incluyen:

- Un variado conjunto de vistas predefinidas que pueden ser usadas también a modo de plantillas para crear distintos componentes en el interfaz de usuario, tales como listas, botones, tablas, e incluso ventanas del navegador embebidas en la aplicación.

- Los *Content Providers*, que permiten que las aplicaciones puedan compartir datos propios entre sí.
- El *Resource Manager*, permite el acceso a recursos tales como etiquetas de idiomas, gráficos, etc.
- El *Notification Manager* permite que cualquier aplicación pueda alertar a los usuarios a través de una notificación en la barra de estado de la pantalla.
- El *Activity Manager*, controla el ciclo de vida de las aplicaciones así como la pila de navegación, esto es, el orden en que las distintas pantallas de las distintas aplicaciones deben ser mostradas en cada momento al usuario.

5. Aplicaciones

Por último, en el nivel más externo de la arquitectura Android encontramos la capa compuesta por las aplicaciones. Estas aplicaciones son tanto las nativas (el calendario, agenda de contactos, navegador, mapas, etc.) como las aplicaciones programadas por cualquier desarrollador.

Se muestra a continuación en esquema con los distintos niveles de la arquitectura Android:

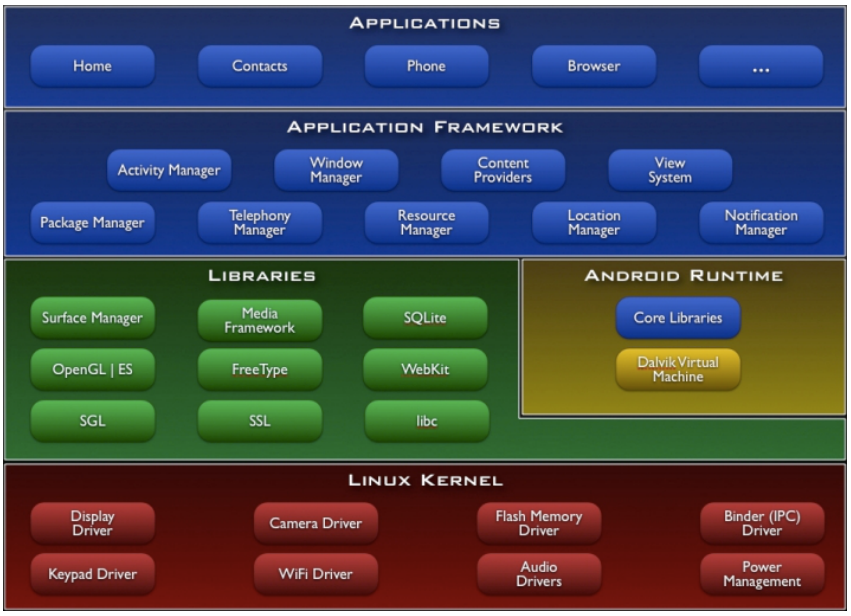

Figura 2.1: Arquitectura Android [17].

2.3. Almacenamiento de datos

En Android hay varias opciones para almacenar datos de una aplicación de forma persistente, dependiendo del grado de privacidad de dichos datos y del espacio que requieran para ser almacenados:

1. Ficheros de preferencias:

Mediante objetos de la clase *SharedPreferences* el programador puede almacenar y recuperar pares de valores de tipo primitivo. Estos valores persistirán entre sesiones de usuario aunque el sistema decida dejar de ejecutar el proceso de la aplicación.

2. Memoria interna del dispositivo:

Los datos almacenados en en la memoria interna son, por defecto, privados a cada aplicación de forma que ni otras aplicaciones ni el usuario tiene acceso a ellos, aunque este nivel de privacidad puede modificarse a través de constantes de forma que los datos sean accesibles por otras aplicaciones. Cuando la aplicación a la que pertenecen se desinstala del dispositivo estos datos son eliminados con ella.

Android permite también guardar datos de forma temporal en la memoria interna usando objetos *File* creados a partir de la función *getCacheDir()*. De esta forma, si el dispositivo necesita tener más espacio disponible, el sistema borra estos ficheros para recuperar espacio. Es importante aún así tener en cuenta que el sistema no se ocupará de borrar estos ficheros por el programador, de forma que éste debe mantener siempre el control sobre ellos y evitar que consuman demasiado espacio. Estos ficheros son eliminados cuando el usuario desinstala la aplicación.

3. Dispositivos externos de almacenamiento:

Todo dispositivo compatible con Android soporta dispositivos compartidos de almacenamiento externo ya sean extraíbles, como por ejemplo las tarjetas SD, o internos de forma que no se pueden separar del dispositivo. Todas las aplicaciones pueden leer y escribir datos de dispositivos externos y el usuario puede borrarlos.

Los datos de una aplicación que sean de uso compartido por varias aplicaciones

deben guardarse en un directorio público del dispositivo de almacenamiento para que de esta forma no sean eliminados cuando el usuario desinstale la aplicación.

Al igual que en el caso de almacenamiento en memoria interna, el desarrollador puede usar archivos para almacenar datos de forma temporal en dispositivos externos, los cuales serán borrados al desinstalar la aplicación. No debe dejarse al sistema la gestión de estos ficheros sino que deben borrarse cuando no sean necesarios y evitar que superen un cierto límite de espacio.

4. Bases de datos:

Android provee soporte a bases de datos SQL, las cuales serán accesibles desde cualquier clase de la aplicación pero no desde fuera de ella. Android no impone ninguna limitación más allá de los conceptos base de SQLite. Es recomendable usar un valor autoincrementado como ID único para cada registro siempre que los datos sean privados; sin embargo, esta recomendación se vuelve obligación cuando se implementa un *content provider*.

4. Conexiones de red:

La última opción que tenemos en Android para almacenar los datos de nuestras aplicaciones es a través de conexiones de red, usando servicios web que almacenen y recuperen los datos.

Android proporciona además un servicio de backup que permite copiar los datos persistentes de una aplicación en un almacén remoto para tener un punto de restauración. En caso de que el usuario efectúe un reset de fábrica o cambie de dispositivo, al instalar de nuevo la aplicación los datos son restaurados de nuevo de forma transparente para él. Sin embargo, debido a que dependiendo del dispositivo y del proveedor del servicio los datos pueden ser almacenados en sitios distintos de la "nube", Android no garantiza la seguridad de esos datos y recomienda ser cautelosos con datos como usuarios y contraseñas.

2.4. Aplicaciones Android

Entre el conjunto de utilidades que están incluidas en el SDK, se encuentra la herramienta aapt que aunque por lo general no sea usada de manera directa por los desarrolladores, es una herramienta empleada por el IDE y los scripts de compilación para generar archivos con formato .apk. Cada uno de estos ficheros constituye una aplicación Android y contiene tanto el código compilado como los ficheros de datos y recursos necesarios para el funcionamiento de la aplicación. Estos son los ficheros usados para distribuir e instalar aplicaciones en esta plataforma.

Cada aplicación Android se ejecuta en un proceso distinto, el cual inicia el sistema operativo cuando es necesario. De la misma forma, es el sistema operativo el que para el proceso cuando ya no es necesario, liberando así los recursos del sistema que son requeridos por otras aplicaciones. Como cada proceso se ejecuta en su propia instancia de la máquina virtual Dalvik, el código de cada aplicación se ejecuta de forma independiente y aislada del resto de aplicaciones [18].

2.4.1. Componentes de una aplicación

La estructura de una aplicación Android está definida por la interacción entre distintos componentes. La aplicación hará uso de las distintas APIs de forma que los componentes encargados de realizar cada tarea puedan ser modificados o reemplazados sin problemas, asegurando así la máxima flexibilidad [13].

A diferencia de otros sistemas, las aplicaciones desarrolladas para esta plataforma no tienen un único punto de entrada a la hora de ejecutar su código, es decir, no tienen un método *main()* por ejemplo. En su lugar, siempre que sea necesario que un componente en concreto maneje alguna petición o evento, el sistema se encarga siempre de que haya:

1. Un proceso de la aplicación ejecutándose, iniciándolo si fuera necesario.

2. Una instancia del componente disponible, creando dicha instancia si fuera necesario.

Existen cuatro tipos de componentes distintos en una aplicación: activities, services, broadcast receivers y content providers.

ACTIVITY

Un componente *activity* representa una actividad realizada por la aplicación y lleva asociada un interfaz de usuario. Cada aplicación puede estar formada por uno o varios de estos componentes, los cuales son independientes entre sí aunque funcionen conjuntamente creando un interfaz de usuario por el que navegar de forma coherente. En cada aplicación se parte de uno de estos componentes para mostrar al usuario, y la forma de pasar de una *activity* a otra es que la primera inicie la segunda.

El conjunto de funcionalidades de la aplicación e interfaces de usuario que compone una *activity* no están predefinidas, sino que es el desarrollador el que decide qué conjunto de operaciones y vistas englobará una determinada *activity*, cada una de ellas implementada como una subclase de la clase base Activity.

A cada *activity* se le asigna una ventana que aunque por defecto se muestre en toda la pantalla, el desarrollador puede adaptar las dimensiones de la misma. Además, cada *activity* puede mostrar ventanas adicionales, tales como ventanas de dialogo emergentes.

Cada ventana contiene una jerarquía de vistas, las cuales son subclases de la clase *View*. Cada vista controla una determinada zona de la pantalla, así como organiza la posición en la que se mostrarán el resto de vistas que descienden de ella. Las vistas del nivel más bajo de la jerarquía son las encargadas de recoger las entradas o respuestas del usuario cuando éste interactúa con las imágenes que quedan dentro de la zona que la vista controla. Android incluye un conjunto de vistas predefinidas que el desarrollador puede usar, tales como botones, barras de scroll, menús, etc.

Ciclo de vida

Estos componentes tienen tres posibles estados:

- Activa: cuando la actividad está en primer plano en pantalla y es el foco de interacción con el usuario.

- Pausada: cuando aún siendo visible, ha perdido el foco porque otra *activity* está por encima de ella, aunque ésta no ocupa toda la pantalla o es transparente. Una *activity* pausada mantiene su estado e información, pero puede ser destruida por el sistema en condiciones extremas de poca memoria libre.

- Parada: cuando la *activity* deja de ser visible porque otra *activity* que hace uso de toda la pantalla pasa a estar en primer plano. La *activity* sigue manteniendo su estado e información, pero el sistema puede terminar su ejecución siempre que se necesite memoria libre.

Cuando una *activity* está pausada o parada, el sistema puede liberar la memoria ocupada por dicha actividad a través de su método *finish()* o bien finalizando su proceso. Es el propio sistema el encargado de restaurar el estado e información de dicha *activity* cuando vuelva a ser necesario.

Las posibles transiciones de un estado a otro en una *activity* se producen mediante llamadas a una serie de métodos, los cuales definen el ciclo de vida de este componente. Es muy importante para todo desarrollador el conocer el estado de una *activity* en cada momento puesto que hay operaciones que deben realizarse en determinados instantes dentro del ciclo de vida de la *activity*, como por ejemplo la liberación de recursos. De esta forma, el ciclo de vida puede dividirse en tres:

1. El ciclo de vida completo está comprendido entre la llamada al método *onCreate()* y la llamada al método *onDestroy()*. Toda la inicialización del estado "global" se hace en el método *onCreate()* y todos los recursos deben ser liberados en el método *onDestroy()*.

2. La parte del ciclo de vida en la que la *activity* está visible es la comprendida entre los métodos *onStart()* y *onStop()*. El usuario podrá visualizar su interfaz gráfico pero sin poder interactuar con él. El programador puede mantener los recursos necesarios para mostrar la *activity* al usuario.

3. La parte del ciclo de vida en que la *activity* está en primer plano, siendo visible de forma que el usuario puede interactuar con ella. Esto se produce entre los métodos *onResume()* y *onPause()*. El código contenido en estos dos métodos debe ser el menor posible, puesto que cuando son invocados se espera una rápida respuesta del sistema en reiniciar/pausar la actividad. En el método *onPause()* se implementan típicamente las operaciones de almacenamiento persistente de datos de forma que no se pierdan en caso de que la actividad finalizara, bien por motivos funcionales de la aplicación o por que el propio sistema terminara la *activity* de forma inesperada.

El desarrollador debe siempre tener presente que el sistema puede terminar una actividad por la necesidad de conservar de memoria y por ello debe guardar el estado dinámico de la *activity*. De esta forma el usuario visualizará sus datos tal y como aparecían previamente a que tuviera lugar tal situación. El desarrollador debe implementar el método *onSaveInstanceState()* para guardar dicho estado dado que Android invoca

este método previamente al método *onPause()* y por tanto a que tenga lugar el fin de la *activity* en cuestión.

A continuación se muestra un diagrama en el que se visualiza con mayor claridad el estado de una *activity* dependiendo del momento en el que se encuentre dentro de su ciclo de vida:

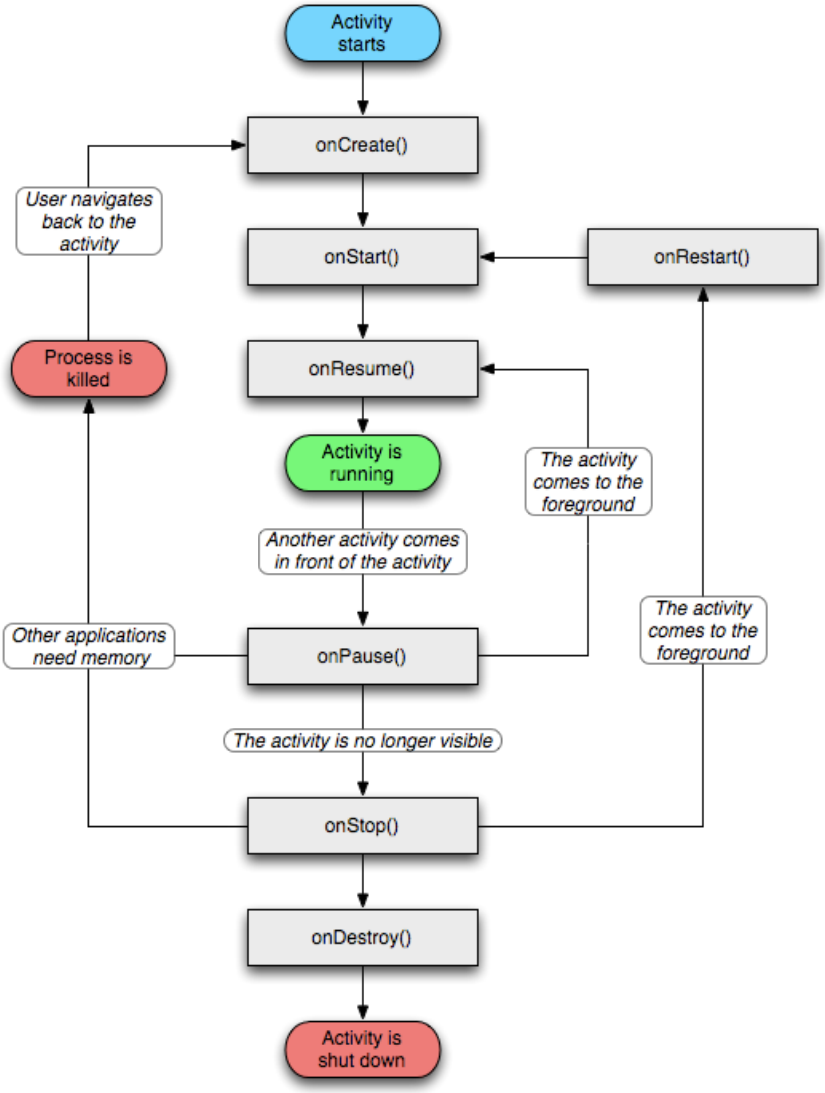

Figura 2.2: Diagrama del ciclo de vida de un componente Activity.

SERVICE

Un componente *service* es una tarea que se ejecuta en segundo plano por un tiempo indefinido. Este tipo de componentes no llevan un interfaz de usuario asociado. Al igual que el componente *activity* y otros componentes, se ejecutan en el hilo principal de la aplicación, de forma que no bloquean otros componentes o el interfaz de usuario. Normalmente estos componentes inician otros hilos para realizar operaciones que conllevan mucho tiempo en completarse. Todo *service* implementado por un desarrollador es una subclase de la clase *Service*.

Es posible la comunicación con un *service* que ya esté ejecutando (o iniciarlo en caso contrario) a través de la interfaz correspondiente.

La mejor forma de entender cómo funciona este tipo de componente es pensar en un reproductor de música. Aunque la aplicación proporcione al usuario pantallas en las que escoger las canciones, ajustar volumen, etc., se espera que la reproducción de la música se mantenga en funcionamiento aunque el usuario pase a realizar otras actividades con su terminal, como por ejemplo leer el correo, navegar por internet, etc.

Ciclo de vida

Un *service* puede ser usado de dos formas:

1. El servicio es iniciado y éste ejecuta hasta que alguien lo para o se para él mismo. En este modo con invocar al método que termina el *service* una vez es suficiente para pararlo, sin importar cuantas llamadas al método que lo inició se hubieran hecho. Los métodos empleados son *Context.startService()*, *Context.stopService()*, *Service.stopSelf()* y *Service.stopSelfResult()*.

2. El *service* puede ser usado mediante un interfaz exportado previamente definido. Múltiples clientes pueden hacer uso de él conectándose al objeto *Service*. Siempre que se use el *service* en este modo, no puede ser terminado mientras haya un cliente conectado al mismo. Los métodos empleados son *Context.bindService()* y *Context.unbindService()*. Hay que tener en cuenta que un *service* iniciado con startService puede recibir conexiones de clientes posteriormente con el método bindService, pero si no estuviera iniciado, este último método se encargaría de hacerlo.

A continuación se muestran dos diagramas del ciclo de vida de un *service*. Se ha hecho esta separación basada en el modo en que se inicia el *service* para una mayor claridad, pero no hay que olvidar lo comentado en el párrafo anterior. En los esquemas se puede

distinguir dos etapas del ciclo de vida:

1. Ciclo de vida entero del *service*, comprendido entre los métodos *onCreate()* y *onDestroy()*.

2. Ciclo de vida activo del *service*, el cual comienza después de la llamada al método *onStart()*.

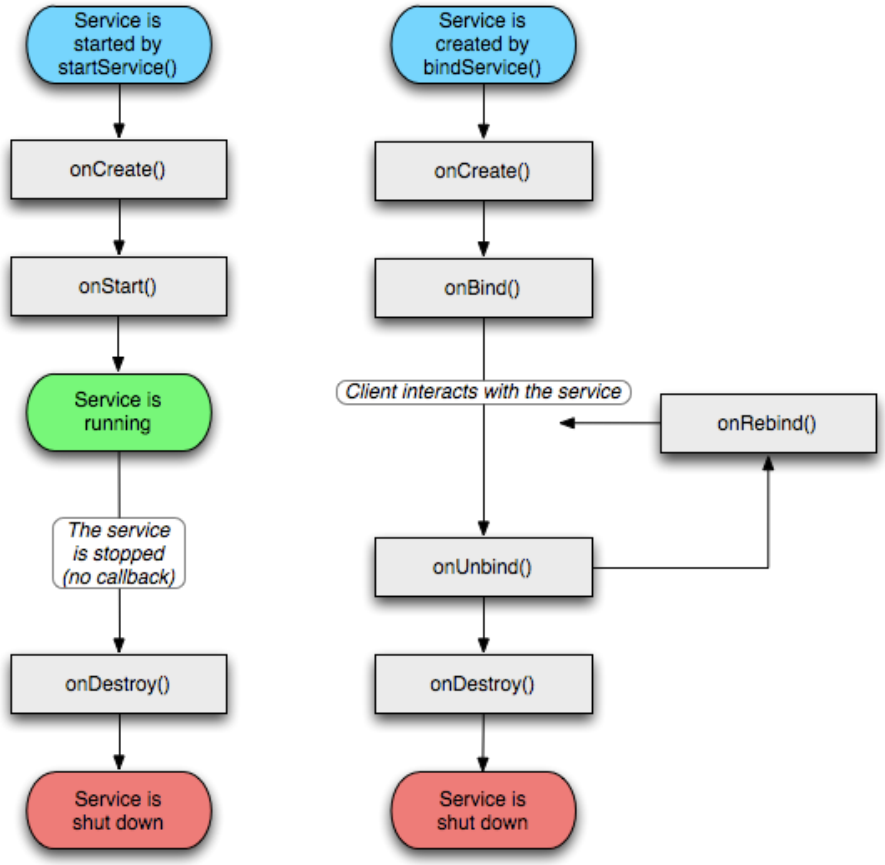

Figura 2.3: Diagrama del ciclo de vida de un componente Service.

BROADCAST RECEIVER

La función de estos componentes es responder mediante una determinada acción a eventos que reciben mediante *broadcast*. Las acciones que se espera realicen depende del evento concreto del que se les notifique, pero puede ser iniciar o terminar una actividad, operaciones en base de datos, notificar al usuario mediante vibración, parpadeo del LED, etc. Aunque un *broadcast receiver* no tiene asociado un interfaz gráfico, puede hacer uso del *Notification Manager* para avisar al usuario del evento producido si fuera necesario. Estos eventos pueden ser lanzados tanto por aplicaciones del sistema como por aplicaciones de terceros. Una aplicación puede utilizar varios de estos componentes, los cuales serán subclases de la clase base *BroadcastReceiver*.

Ciclo de vida

Un componente de este tipo sólo está activo mientras se ejecuta su método *onReceive()*. Si un *broadcast receiver* inicia un nuevo hilo y después termina la ejecución del método *onReceive()*, existe la posibilidad de que Android termine el proceso en que se producen estas ejecuciones si ningún otro componente permanece activo, de forma que las operaciones a realizar por el hilo iniciado no llegan a completarse. El desarrollador debe tener en cuenta esta situación y asignar las operaciones a realizar a un componente *service* iniciado por el *broadcast receiver*. De esta forma, al estar el componente *service* activo, el sistema no terminará el proceso en que se ejecuta.

CONTENT PROVIDERS

La función de este tipo de componente consiste en hacer posible que distintas aplicaciones puedan compartir determinados conjuntos de datos entre sí a través de una base de datos SQLite, ficheros de sistema, etc. Son subclases de la clase *ContentProvider*, la cual posee métodos que permiten a las aplicaciones almacenar y recuperar datos del tipo concreto que controlan. Sin embargo, las aplicaciones no se comunican directamente con este tipo de objetos, sino con otros de tipo *ContentResolver*, los cuales hacen las llamadas a los métodos mencionados anteriormente.

Ciclo de vida

Un *content provider* está activo únicamente mientras responde a una petición de un *ContentResolver*, por lo que el programador no necesita estar pendiente de terminar su ejecución.

2.4.2. Activación de componentes

Mientras que los *content providers* se activan a través de un *ContentResolver*, los otros tres tipos de componentes son activados mediante *intents*, que son mensajes asíncronos implementados por objetos del tipo *Intent*. Un *intent* en sí es una estructura de datos que porta información sobre la acción a realizar y los datos sobre los que actuar en el caso de ir dirigido a una *activity* o a un *service*. En el caso de ir dirigido a un *broadcast receiver* la información que porta es sobre una acción que ha tenido lugar y que hay que notificar.

El mecanismo para activar los tres tipos de componentes es distinto en cada caso, de forma que el sistema dirige el intent al componente adecuado en cada caso llegándolo a instanciar si fuera necesario.

La información que puede contener un *intent* es:

- **Nombre del componente:** Este campo se refiere al nombre del componente que debe manejar el *intent*. Es un campo opcional que en caso de ir informado el *intent* se dirige a una instancia de la clase designada. En caso contrario, el sistema se encarga de encontrar otro componente adecuado al que dirigir el mensaje.

- **Acción:** Es una cadena de texto referida a la acción a realizar o notificar. La clase *Intent* proporciona una serie de constantes ya definidas, aunque el desarrollador puede definir sus propias acciones también. Este campo determina la forma en la que el resto del *intent* es estructurado.

- **Datos:** La URI de los datos sobre los que ejecutar la acción y el tipo MIME de dichos datos. Los valores de este campo serán acordes al valor de la acción definida. Es decir, si la acción es ACTION_EDIT este campo contendría la ruta del documento a editar. En cambio, si la acción fuera ACTION_CALL, el valor contenido en los datos sería un número telefónico al que llamar.

- **Categoría:** Contiene información adicional sobre el tipo de componente a activar en forma de cadena de texto. Se pueden incluir varias categorías en un mismo *intent*. Al igual que con las acciones, la clase *Intent* provee categorías predefinidas.

- **Extras:** Son pares clave-valor de información adicional que enviar al componente que debe ser activado. Este campo también depende de la acción definida. Por ejemplo, si la acción fuera SHOW_COLOR, el valor del color debería ser especificado en un par clave-valor.

- **Flags:** Son *flags* de varios tipos para indicar como lanzar una actividad y cómo actuar una vez ésta ha sido iniciada. Estos *flags* están definidos en la clase *Intent*

Los *intents* pueden ser explícitos o implícitos. Un *intent* es explícito cuando lleva informado el nombre del componente a activar, de forma que irá dirigido a una instancia de esa clase sin importar el resto de campos de la estructura. En cambio, los *intents* implícitos no llevan definidos un nombre de componente. Es en estos casos cuando el sistema hace uso de filtros (instancias de la clase *IntentFilter*) para determinar qué componente debe manejar el mensaje. Si un componente no tiene *intent filters* sólo puede recibir *intents* explícitos [19].

2.4.3. Recursos de una aplicación

A la hora de desarrollar una aplicación siempre es una ventaja independizar los recursos, tales como imágenes o cadenas de texto, del código en sí. Esto permite hacer un mejor mantenimiento de la aplicación frente a posibles cambios en dichos recursos, además de hacerla compatible con un mayor número de configuraciones en los dispositivos, tales como diferentes idiomas o tamaños de pantalla. Para ello, los recursos deben ser organizados en el directorio `res/` del proyecto usando varios subdirectorios que los agrupen según tipo y configuración.

Para cada tipo de recurso se pueden definir recursos por defecto y recursos alternativos.

- Recursos por defecto: Son los empleados por Android sin tener en cuenta la configuración del dispositivo o cuando no hay recursos alternativos definidos para la configuración dada.
- Recursos alternativos: Son los definidos para ser usados con configuraciones específicas. Los subdirectorios de recursos alternativos tienen nombres específicos que Android reconoce, de forma que automáticamente aplica los recursos apropiados para la configuración dada.

Los posibles directorios en los que agrupar los tipos de recursos son:

- `anim/:` Ficheros .xml que definen animaciones.
- `color/:` Ficheros .xml que definen listas de estado de colores, las cuales definen colores que cambian según el estado de una vista.

- `drawable/`: Imágenes (.png, .9.png, .jpg, .gif) definidas como bitmaps o como .xml.
- `layout/`: Ficheros .xml que definen interfaces de usuario.
- `menu/`: Ficheros .xml que definen menús de la aplicación.
- `raw/`: Los ficheros se guardan en este directorio en forma de fila y no son comprimidos por el sistema.
- `values/`: Ficheros .xml que contienen recursos simples como arrays, colores, tamaños, texto o estilos.
- `xml/`: Ficheros .xml que pueden ser leídos en tiempo de ejecución.

La forma de definir recursos alternativos es añadir sufijos determinados, separados por guiones, a los subdirectorios anteriormente comentados. Por ejemplo, para definir recursos específicos a cada tamaño de pantalla, se añadirán los valores `small, normal o large` . Si se quieren especificar recursos para un determinado idioma, se añadirá el sufijo de ese idioma. Cuando se requiere añadir varios sufijos para especificar los recursos según varias características de la configuración, se hará siguiendo unas determinadas reglas predefinidas en la documentación de la plataforma [20].

2.4.4. El archivo AndroidManifest.xml

Todas las aplicaciones Android deben contener un archivo AndroidManifest.xml en su directorio raíz. Este fichero tiene información esencial que el sistema necesita conocer antes de ejecutar el código. Algunas de sus funciones son [21]:

- Declarar el nombre del paquete Java que da nombre a la aplicación y que sirve de identificador único para la misma.
- Describe los componentes de los que se compone la aplicación, así como las clases que implementan dichos componentes y sus capacidades, éstas últimas definidas mediante los filtros ya comentados en secciones anteriores. Estas declaraciones permiten al sistema saber qué componentes existen y cuales son las condiciones en las que pueden iniciarse.
- Determina qué procesos gestionarán los componentes de la aplicación.
- Declara los permisos que la aplicación necesita para acceder a partes del API que son protegidas. También declara los permisos necesarios tanto para interactuar con otras aplicaciones como para que terceros interactúen con los componentes de la aplicación dada.

- Declara el nivel minimo del API de Android que la aplicación necesita para ejecutarse.
- Lista las librerías de las que la aplicación hace uso.

2.5. Seguridad y permisos

En todo desarrollo una de las partes más importantes a tener en cuenta es la seguridad, especialmente cuando la aplicación maneja datos que pueden ser de carácter confidencial o privado. Aprovechando las características que ofrece Linux, gran parte de la seguridad de Android está determinada a nivel de proceso. Adicionalmente, Android provee un sistema de permisos que imponen restricciones sobre ciertas operaciones que los procesos pueden realizar. Dichos permisos pueden ser tanto los definidos por el propio sistema Android, como permisos definidos por las propias aplicaciones.

Android ha sido diseñado de forma que por defecto las aplicaciones no tienen permiso para realizar cualquier operación que pudiera causar algún tipo de impacto sobre otras aplicaciones, el sistema operativo o el propio usuario. Como se ha explicado ya, los permisos requeridos por la aplicación son declarados en el archivo AndroidManifest.xml. De esta forma, el usuario puede conocer los permisos necesarios de forma previa a la instalación de la aplicación, pudiendo decidir si continuar con la misma o no.

Otra característica relacionada con la seguridad es el hecho de que toda aplicación Android debe ser firmada por el desarrollador con un certificado para poder ser instalada. La única función del certificado es establecer relaciones de confianza entre aplicaciones. Gracias a las firmas se puede determinar quien puede acceder a permisos basados en firma y quien puede compartir los identificadores de usuario.

Cuando una aplicación es instalada en un dispositivo se le asigna un único identificador de usuario de Linux, el cual será siempre el mismo mientras la aplicación esté instalada. Esto hace a cada aplicación ejecutarse en un proceso independiente del resto de forma que no puedan interactuar entre sí. El atributo *sharedUserId* del elemento *manifest* en el fichero AndroidManifest.xml consigue que a dos paquetes distintos se les asigne el mismo ID de usuario de forma que son tratados como si fueran una única aplicación y por tanto comparten permisos. Aún así, para mantener un cierto nivel de seguridad, sólo dos aplicaciones firmadas con el mismo certificado pueden tener el mismo ID de usuario.

Los datos almacenados por una aplicación son asociados al ID de usuario de dicha aplicación, de forma que sólo la misma tiene acceso a ellos. Sin embargo, existen parametros para hacer que los datos sean accesibles en modo lectura y escritura por otras aplicaciones, aunque sigan perteneciendo a la primera.

Por último, añadir que los permisos también pueden ser otorgados a nivel de las URI que manejan los *ContentProvider* [22].

2.6. Localización y mapas

Dado que la geolocalización es una de las principales funcionalidades de este proyecto, a continuación se hará una breve explicación sobre las capacidades que ofrece Android para usar este tipo de servicios.

La localización en aplicaciones móviles se refiere a la capacidad de conocer datos tales como las coordenadas geográficas, altitud, rumbo, etc., de la localización de un dispositivo, y por tanto del usuario que lo porta. Android provee acceso a los servicios de localización que ofrece el dispositivo a través del paquete `android.location` . El componente principal del mismo es el componente *LocationManager* del sistema, el cual provee un API para conocer los datos de geolocalización deseados.

Mediante el *LocationManager* se puede obtener una lista de proveedores disponibles para la última posición conocida del usuario. Además, se pueden obtener actualizaciones periódicas sobre la localización del dispositivo, así como programar avisos cuando el dispostivo se encuentre cercano a unas determinadas coordenadas previamente definidas.

Una funcionalidad que complementa la anteriormente comentada es la visualización de mapas en el dispositivo, donde se pueden mostrar localizaciones de interés para el usuario. Google proporciona la librería externa `com.google.android.maps` para tal fin. Esta librería por tanto no viene incorporada en el SDK de la plataforma, sino que hay que añiadirla a posteriori al proyecto y declararla en el archivo AndroidManifest.xml. La clase principal de esta librería es *MapView*, la cual muestra mapas obtenidos a través del servicio Google Maps junto con controles y funciones para que el usuario maneje el mapa. Para que los mapas de Google Maps puedan ser visualizados en esta vista es imprescindible registrarse en el servicio para obtener una clave única [23]que incluir en el código [24].

2.7. Herramientas

El SDK de Android viene con una serie de herramientas incluidas, algunas de las cuales son de uso indispensable a la hora de desarrollar una aplicación. Dependiendo del entorno de desarrollo, el uso de alguna de ellas por parte del propio programador ni siquiera será necesario ya que será el IDE el que las utilice. A continuación se explicarán dos de las herramientas más usadas, incluso necesarias, en el desarrollo de una aplicación:

AVDs (Android Virtual Devices)

- Cada AVD no es más que una configuración determinada para el emulador, pudiendo crear configuraciones lo más parecidas posibles a dispositivos reales. Están compuestos de:

- Un perfil hardware: Se pueden establecer opciones para definir las características del hardware del dispositivo virtual, como por ejemplo el tipo de teclado o el tamaño de pantalla.
- Una asignación de una imagen del sistema. Se puede especificar qué versión de la plataforma se desea que ejecute el emulador.
- Otras opciones: Es posible especificar la apariencia, tamaño de pantalla, tarjeta SD, etc.
- Una zona en la máquina de desarrollo dedicada a almacenar los datos del usuario y la tarjeta SD simulada.

El desarrollador puede crear tantos AVDs como necesite, pero siempre tiene que existir uno asociado al proyecto que se desee ejecutar en el emulador. La plataforma proporciona una herramienta para el manejo de los AVDs, la cual puede usarse tanto a través de una interfaz gráfica como por linea de comandos [25].

DDMS (Dalvid Debug Monitor Service)

- El DDMS es una herramienta que incorpora Android la cual proporciona muchas facilidades al desarrollador a la hora de depurar una aplicación. Algunas de las posibilidades que ofrece son: capturas de pantalla, información sobre hilos, procesos y memoria, estado de la radio, simulación de llamadas, SMS y localización, etc. El DDMS funciona tanto con el emulador como con dispositivos reales.

El DDMS incorpora un grupo de controles con los que poder simular estados y acciones especiales del dispositivo, los cuales se encuentran en la pestaña llamada "Controles del emulador.Esta herramienta es indispensable cuando la aplicación a desarrollar incorpora funcionalidades de geolocalización y mapas.

- Se puede cambiar el plan de voz y datos del dispositivo, así como el tipo de red en la que opera.
- Se pueden simular llamadas entrantes así como SMSs.
- Se puede simular la localización GPS del dispositivo. Para realizar esta operación hay tres opciones: introducciendo las coordenadas manualmente, a través de ficheros .gpx o mediante ficheros .kml.

La herramienta muestra los emuladores/dispositivos encontrados con una lista de todas las VMs que se ejecutan en cada uno de ellos. También es posible visualizar una lista de procesos que corren en cada VM. La información que en la pestaña de hilos se muestra es:

- **ID** - Identificador único asignado a una VM.
- **Tid** - El ID del hilo, que para el hilo principal del proceso será el mismo que el de el proceso en sí.
- **Status** - El estado del hilo de la VM, que puede ser uno de los siguientes:
  - running - Cuando ejecuta el código de la aplicación.
  - sleeping - Cuando se ha invocado el método *Thread.sleep()*.
  - monitor - Cuando está esperando para el bloqueo de un monitor.
  - wait - Se inicia después de la invocación de *Object.wait()*.
  - native - El hilo está ejecutando código nativo.
  - wmwait - Cuando se está esperando por un recurso de la VM.
  - zombie - El hilo está en proceso de finalización.
  - init - El hilo se está inicializando.
  - starting - El hilo está apunto de comenzar su ejecución.
- **utime** - Tiempo total empleado en ejecutar código de usuario en unidades de tiempo (normalmente 10 ms).
- **stime** - Tiempo total empleado en ejecutar código del sistema en unidades de tiempo (normalmente 10 ms).

- **Name** - Nombre del hilo.

En cuanto a esta herramienta, cabe también destacar la utilidad del explorador de archivos. El desarrollador puede navegar a través de los directorios y conocer el estado del sistema. Además puede mover archivos dentro de un mismo directorio, borrarlos y exportar e importarlos. Un uso muy práctico que tiene esta última funcionalidad es poder extraer del dispositivo la base de datos de forma que se pueden conocer sus datos en cualquier momento [26].

Capítulo 3

OTRAS HERRAMIENTAS UTILIZADAS

3.1. Servidor Tomcat

Tanto los servicios web como las dos aplicaciones web implementadas en este proyecto se han desplegado en el servidor Tomcat por ser una solución sin coste de licencia y de fácil mantenimiento, entre otras características que se analizarán posteriormente.

3.1.1. Introducción y características

Tomcat, también conocido como Apache Tomcat o Jakarta Tomcat, es un proyecto que comenzó a desarrollarse en Sun Microsystems. Posteriormente, el código base fue donado a la *Apache Software Foundation* y desde entonces múltiples voluntarios de Sun y otras organizaciones han contribuido a su desarrollo.

Tomcat es una implementación *opensource* completamente funcional de los estándares de *Java Servlets* y *JavaServer Pages*. Incluye el compilador Jasper, que compila JSPs convirtiéndolas en *servlets*. También puede especificarse como el manejador de las peticiones de JSP y *servlets* recibidas por servidores web como por ejemplo el servidor Apache HTTP de la *Apache Software Foundation*. Tomcat está integrado en la implementación de referencia Java 2 Enterprise Edition (J2EE) de Sun Microsystems.

Aunque Tomcat también es conocido como Apache Tomcat, no depende del servidor

web Apache para su funcionamiento. Apache es una implementación en C de un servidor web HTTP que es capaz de servir páginas HTML. Aunque en sus inicios existía la percepción de que el uso de Tomcat de forma autónoma sólo era recomendable para entornos de desarrollo y entornos con requisitos mínimos de velocidad y gestión de transacciones, actualmente la realidad es que Tomcat se puede usar como servidor web autónomo en entornos con altos nivels de tráfico y alta disponibilidad [27].

Una de las ventajas de este servidor es que fue escrito en Java y, por tanto, funciona en cualquier sistema operativo que disponga de la máquina virtual de Java.

3.1.2. Estructura de directorios de una aplicación

Dentro del Tomcat, pueden encontrarse los siguientes directorios:

- */bin*: arranque, cierre, y otros scripts y ejecutables.
- */temp*: archivos temporales.
- */conf*: archivos XML y los correspondientes DTD para la configuración de Tomcat. El mas importante es server.xml, del que se hablará posteriormente.
- */logs*: archivos de registro (log) del servidor.
- */webapps*: directorio que contiene las aplicaciones web.
- */work*: almacenamiento temporal de ficheros y directorios.

Como se acaba de mostrar, las aplicaciones desplegadas en el servidor Tomcat se pueden encontrar en la carpeta *webapps* del mismo. La estructura básica de directorios para cada aplicación es la siguiente:

- Nombre-de-la-aplicación : Es el directorio raíz de la aplicación. Contiene la parte púplica de la misma, es decir, los archivos .jsp, imágenes, ficheros *javascript*, etc. Todos estos tipos de archivos pueden agruparse en diferentes subdirectorios para una mejor organización.
  - META-INF: Directorio opcional, suele contener el archivo MANIFEST.MF, el cual indica lass bibliotecas de las que depende la aplicación.
  - WEB-INF:
    - classes: Este directorio contiene las clases java compiladas.

- src: Directorio opcional para contener el código fuente de las clases .java.
- lib: Contiene las librerías necesarias para el correcto funcionamiento de la aplicación.
- web.xml: Es el descriptor de despliegue de la aplicación.

Esta estructura de directorios está contenida en el archivo WAR (Web Archive). Estos archivos definidos por Sun deben estar presentes en cualquier producto de "Servlet Engine"(Contenedor-Web). El archivo WAR en sí no es legible sino que tiene que ser expandido/descomprimido para ser leído. Cuando se lleva acabo la ejecución de Tomcat éste inspecciona y automáticamente expande cualquier archivo WAR que se encuentra bajo el directorio *webapps*.

3.1.3. Configuración del servidor Tomcat

La configuración de Tomcat se basa en dos ficheros:

1. server.xml - El fichero de configuración global de Tomcat.
2. web.xml - Configura los distintos contextos en Tomcat.

server.xml

El fichero *server.xml* tiene dos objetivos [28]:

1. Proporcionar configuración inicial para los componentes de Tomcat.
2. Especifica la estructura de Tomcat, lo que significa, permitir que Tomcat arranque y se construya a sí mismo mediante los componentes especificados en server.xml.

Los elementos más importantes de este fichero son:

- **Server**: El elemento superior del fichero *server.xml*. Server define un servidor Tomcat y puede contener elementos *Logger* y *ContextManager*.
- **Logger**: Este elemento define un objeto *logger*. Cada objeto de este tipo tiene un nombre que lo identifica, así como una ruta para el fichero *log* que contiene la salida y un *verbosityLevel*, el cual especifica el nivel de *log*.

- **ContextManager**: Especifica la configuración y la estructura para un conjunto de *ContextInterceptors, RequestInterceptors, Contexts* y sus *Connectors*.

- **ContextInterceptor** y **RequestInterceptor**: Estos interceptores escuchan ciertos eventos que sucenden en el *ContextManager*. Por ejemplo, el *ContextInterceptor* escucha los eventos de arrancada y parada de Tomcat, y *RequestInterceptor* mira las distintas fases por las que las peticiones de usuario necesitan pasar durante su servicio.

- **Connector**: El *Connecto* representa una conexión al usuario, a través de un servidor web o directamente al navegador del usuario. El objeto *connector* es el responsable del control de los *threads* en Tomcat y de leer/escribir las peticiones/respuestas desde los *sockets* conectados a los distintos clientes. La configuración de los conectores incluye información como: la clase manejadora, el puerto TCP/IP donde escucha el controlador y el *backlog* TCP/IP para el *server socket* del controlador.

- **Context**: Representa una ruta en el árbol del servidor donde situanos una aplicación web.

web.xml

El archivo *web.xml* es el descriptor de despliegue de la aplicación y algunos de sus parámetros son:

- **display-name**: define el nombre de la aplicación web.

- **description**: define la descripción de la aplicación web.

- **listener**: identifica la clase Java que se invocará durante el arranque y parada de la aplicación.

- **welcome-file-list**: define el punto de entrada a la aplicación.

- **servlet**: declaración de un servlet.

- **servlet-mapping**: mapeo de un servlet a una URL.

- **session-config**: permite configurar parámetros de la sesión.

3.1.4. Integración en el IDE Eclipse

Antiguamente si se quería integrar Tomcat en Eclipse era necesario recurrir a algún *plugin*, como *Sysdeo Eclipse Tomcat Launcher* [29]. Sin embargo, desde que se lanzara Eclipse Ganymede (Eclipse 3.4) eso ya no es necesario y es posible iniciar y parar Tomcat o depurar el código desde Eclipse. En concreto, todo el código desarrollado para este proyecto se ha implementado usando Eclipse, de forma que se ha centralizado toda la actividad en el mismo entorno con la rapidez y comodidad que ello conlleva.

Conseguir poder trabajar con Tomcat embebido dentro de Eclipse es, además, muy sencillo. Para ello los pasos a seguir son:

1. Descargar Eclipse desde su página web y descomprimir el .zip de la última versión *core* de Tomcat.

2. Iniciar Eclipse y dirigirse a "Window" - "Preferences" - "Server" - "Runtime Environment", pulsar "Add" y seleccionar la versión de Tomcat descargada. Marcar "Create a new local server" si no está seleccionado.

3. Pulsar "Next" y buscar el directorio en el que se instaló Tomcat. Pulsar "Finish" y "OK". El nuevo servidor debería mostrarse en la pestaña "Servers" 3.2. También se habrá creado un proyecto nuevo "Servers" con los archivos de configuración de nuestra instancia de Tomcat 3.1.

Para desplegar una aplicación en el servidor, basta con arrastrarla desde la pestaña de exploración de paquetes hasta el servidor en la pestaña de Servidores, donde aparecen los controles para iniciar, parar, depurar, etc. el Tomcat.

Otras de las funciones muy útiles que proporciona Eclipse es la exportación un proyecto de una aplicación a un fichero .war para su despliegue en otro servidor. Esta opción está disponible a través del menú contextual que aparece al pulsar sobre el proyecto con el botón derecho del ratón.

Por último, comentar que Eclipse no despliega las aplicaciones en el directorio donde se instaló originalmente el servidor Tomcat, sino que las despliega en un subdirectorio del *workspace*. Éste directorio es :

..\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp1\wtpwebapps

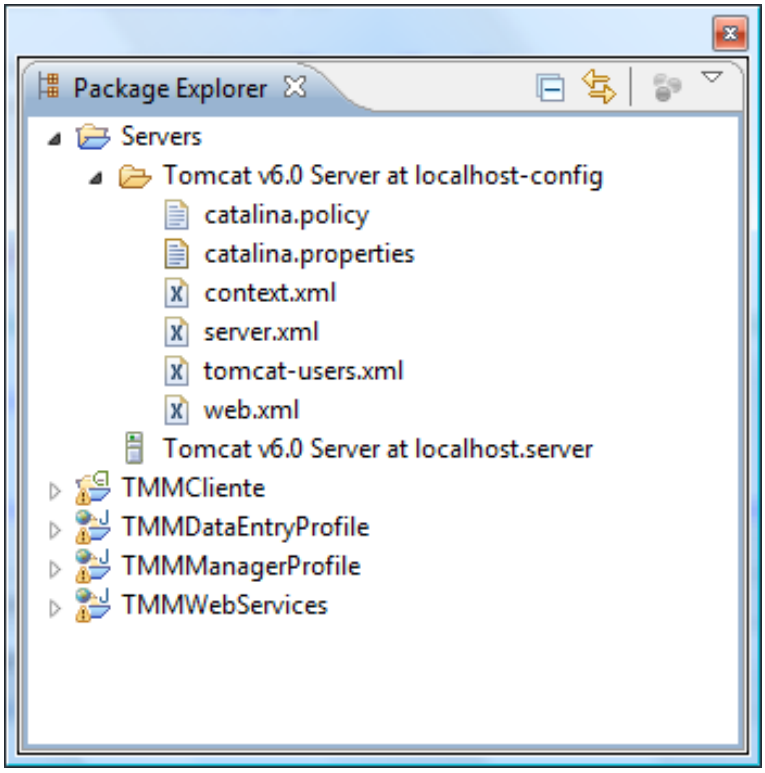

Figura 3.1: Proyecto *Servers* en la pestaña de exploración de Eclipse.

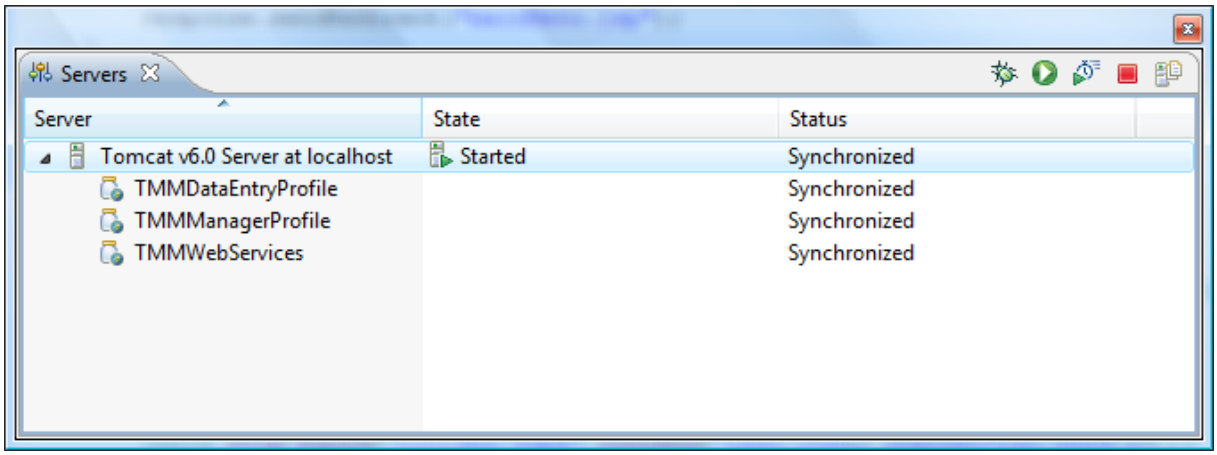

Figura 3.2: Proyecto *Servers* en la pestaña "Servers" de Eclipse.

3.2. Base de datos MySQL

En esta sección se explicará el gestor de base de datos MySQL. Previamente a ello, se hará una exposición breve sobre los aspectos más importantes de las bases de datos relacionales, por ser de este tipo la herramienta MySQL.

3.2.1. Introducción a las bases de datos relacionales

Una base de datos relacional es un conjunto de datos que están almacenados en tablas entre las cuales se establecen unas relaciones para manejar los datos de una forma eficiente y segura. Para usar y gestionar una base de datos relacional se usa el lenguaje estándar de programación SQL.

Edgar Frank Codd definió las bases del modelo relacional a finales de los 60. Trabajaba para IBM, empresa que tardó un poco en implementar sus bases. Pocos años después el modelo se empezó a implementar cada vez más, hasta ser el modelo de bases de datos más popular. En las bases de Codd se definían los objetivos de este modelo [30]:

- **Independencia física**. La forma de almacenar los datos, no debe influir en su manipulación lógica.
- **Independencia lógica**. Las aplicaciones que utilizan la base de datos no deben ser modificadas por que se modifiquen elementos de la base de datos.
- **Flexibilidad**. La base de datos ofrece fácilmente distintas vistas en función de los usuarios y aplicaciones.
- **Uniformidad**. Las estructuras lógicas siempre tienen una única forma conceptual(las tablas)
- **Sencillez**.

Como se acaba de explicar, las bases de datos relacionales se basan en el uso de tablas, también llamadas relaciones. Las tablas se representan gráficamente como una estructura rectangular formada por filas y columnas. Cada columna almacena información sobre una propiedad determinada de la tabla. Cada fila posee una ocurrencia o ejemplar de la instancia o relación representada por la tabla. El tipo de tablas existentes en este modelo, son las siguientes:

- **Persistentes**: Únicamente pueden ser borradas por los usuarios.

- **Base**: Independientes, se crean indicando su estructura y sus ejemplares.
- **Vistas**: Son tablas que sólo almacenan una definición de consulta, resultado de la cual se produce una tabla cuyos datos proceden de las bases o de otras vistas e instantáneas. Si los datos de las tablas base cambian, los de la vista que utiliza esos datos también cambia.
- **Instantáneas**. Son vistas que sí almacenan los datos que muestra, además de la consulta que dio lugar a esa vista. Únicamente actualizan los datos siendo refrescadas por el sistema cada cierto tiempo.
- **Temporales**. Son tablas que se eliminan automáticamente por el sistema. Pueden ser de cualquiera de los tipos anteriores.

A continuación se presenta la terminología relacional:

- **Tupla**: Cada fila de la tabla (cada ejemplar que la tabla representa).
- **Atributo**: Cada columna de la tabla.
- **Grado**: Número de atributos de la tabla.
- **Cardinalidad**: Número de tuplas de una tabla.
- **Dominio**: Conjunto válido de valores representables por un atributo. Los dominios suponen una gran mejora en este modelo ya que permiten especificar los posibles valores válidos para un atributo. Cada dominio incorpora su nombre y una definición del mismo.

En cuanto a las claves de las tablas, hay varios tipos:

- **Clave candidata**: conjunto de atributos de una tabla que identifican unívocamente cada tupla de la tabla.
- **Clave primaria**: clave candidata que se escoge como identificador de las tuplas.
- **Clave alternativa**: cualquier clave candidata que no sea primaria.
- **Clave externa**: atributo de una tabla relacionado con una clave de otra tabla.

Las restricciones son unas condiciones de obligado cumplimiento por los datos de la base de datos. Éstas pueden ser:

1. **Inherentes**: Son aquellas que no son determinadas por los usuarios, sino que son definidas por el hecho de que la base de datos sea relacional. Por ejemplo:
   - No puede haber dos tuplas iguales.
   - El orden de las tuplas no importa.
   - El orden de los atributos no importa.
   - Cada atributo sólo puede tomar un valor en el dominio en el que está inscrito.
2. **Semánticas**: El modelo relacional permite a los usuario incorporar restricciones personales a los datos. Las principales son:
   - **Clave primaria**: Hace que los atributos marcados como clave primaria no puedan repetir valores.
   - **Unicidad**: Impide que los valores de los atributos marcados de esa forma puedan repetirse.
   - **Obligatoriedad**: Prohibe que el atributo marcado de esta forma no tenga ningún valor.
   - **Integridad referencial**: Prohibe colocar valores en una clave externa que no estén reflejados en la tabla donde ese atributo es clave primaria.
   - **Regla de validación**: Condición que debe de cumplir un dato concreto para que sea actualizado.

3.2.2. MySQL

MySQL es un sistema de gestión de bases de datos relacionales, licenciada bajo la GPL de la GNU. Su diseño multihilo le permite soportar una gran carga de forma muy eficiente. MySQL fue creado por la empresa sueca MySQL AB, aunque fue adquirido por Sun Microsystems en el año 2008. MySQL surgió como un intento de conectar el gestor mSQL a las tablas propias de MySQL AB, usando sus propias rutinas a bajo nivel. Tras unas primeras pruebas, vieron que mSQL no era lo bastante flexible para lo que necesitaban, por lo que tuvieron que desarrollar nuevas funciones. Esto resultó en una interfaz SQL a su base de datos, con una interfaz totalmente compatible a mSQL.

Aunque MySQL es *software libre*, se distribuye una versión comercial de MySQL, que no se diferencia de la versión libre m's que en el soporte técnico que se ofrece, y la posibilidad de integrar este gestor en un software propietario, ya que de no ser así, se vulneraría la licencia GPL.

Este gestor de bases de datos es, probablemente, el gestor más usado en el mundo del *software* libre, debido a su gran rapidez y facilidad de uso. Esta gran aceptación es debida, en parte, a que existen infinidad de librerías y otras herramientas que permiten su uso a través de gran cantidad de lenguajes de programación, además de su fácil instalación y configuración.

Inicialmente, MySQL carecía de algunos elementos esenciales en las bases de datos relacionales, tales como integridad referencial y transacciones. A pesar de esto, atrajo a los desarrolladores debido a su simplicidad, de tal manera que los elementos faltantes fueron complementados por la vía de las aplicaciones que la utilizan. Poco a poco estos elementos faltantes, están siendo incorporados tanto por desarrolladores internos, como por desarrolladores de *software* libre. Se pueden destacar las siguientes características principales [31]:

- El principal objetivo de MySQL es velocidad y robustez.
- Soporta gran cantidad de tipos de datos para las columnas.
- Gran portabilidad entre sistemas, puede trabajar en distintas plataformas y sistemas operativos.
- Cada base de datos cuenta con 3 archivos: Uno de estructura, uno de datos y uno de índice y soporta hasta 32 índices por tabla.
- Aprovecha la potencia de sistemas multiproceso, gracias a su implementación multihilo.
- Flexible sistema de contraseñas y gestión de usuarios, con un muy buen nivel de seguridad en los datos.
- El servidor soporta mensajes de error en distintas lenguas.

En la figura 3.3 se muestra un diagrama de la arquitectura de MySQL.

Una vez que se ha explicado el gestor MySQL y se han expuesto sus características, se pueden citar a continuación varias razones por las que utilizar MySQL como gestor para la base de datos en este proyecto, teniendo en cuenta que está enfocado a un entorno empresarial [32].

1. **Escalabilidad y flexibilidad**: ofrece lo último en escalabilidad, siendo capaz de manejar bases de datos empotradas ocupando sólo 1MB, y hacer funcionar sistemas de incluso *terabytes* de información. La flexibilidad de plataforma es una

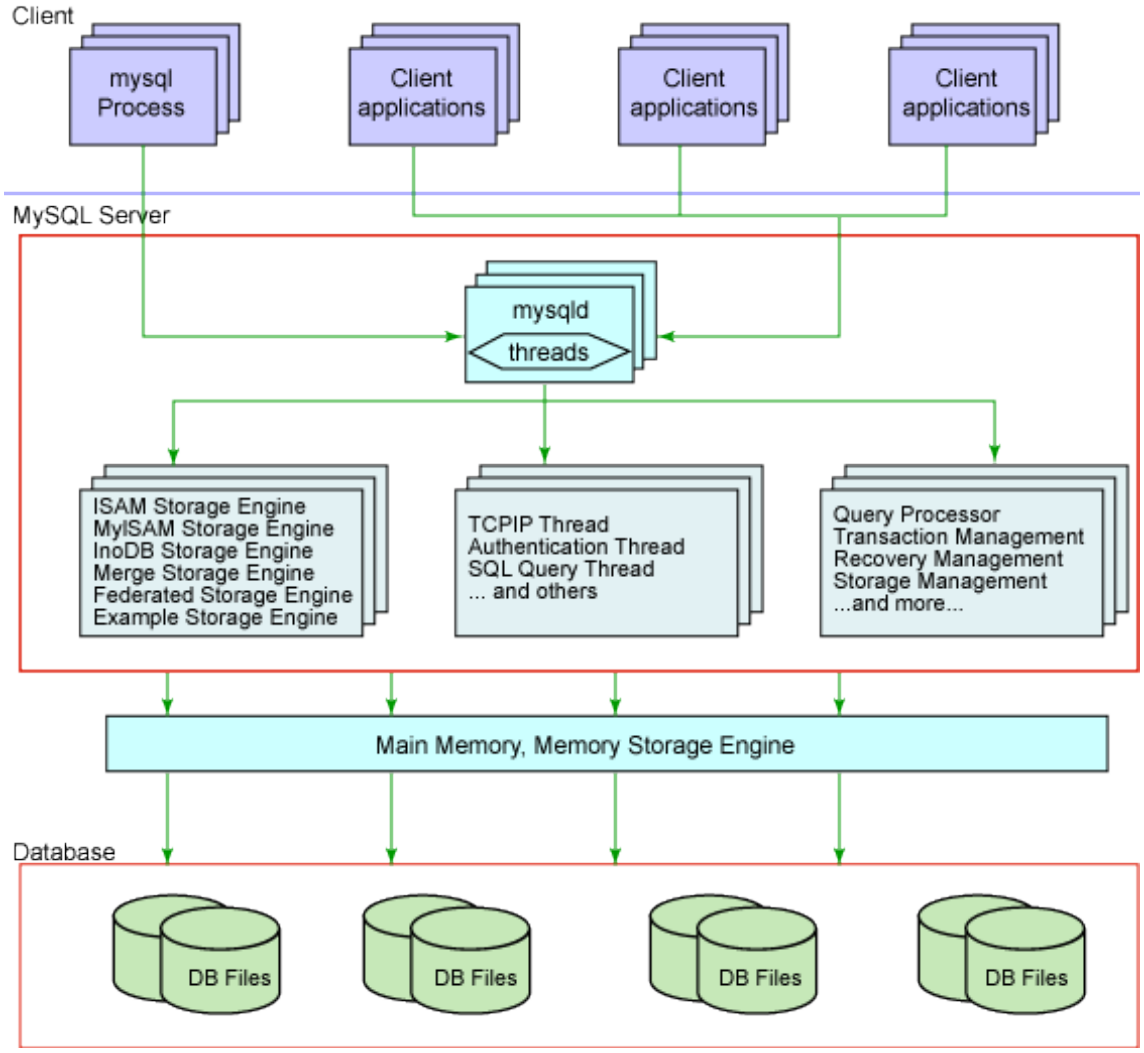

Figura 3.3: Arquitectura de MySQL.

característica clásica de MySQL, soportando distintas versiones de Linux, UNIX y Windows. La naturaleza *open source* de MySQL permite una personalización completa para aquellos que deseen añadir características al servidor.

2. **Alto rendimiento**: su arquitectura permite a los profesionales configurar el servidor MySQL para aplicaciones específicas, dando como resultado un rendimiento espectacular. MySQL puede cumplir con las expectaciones de rendimiento de cualquier sistema, ya sea un sistema de procesamiento transaccional de alta velocidad, o un sitio web de gran volúmen. MySQL es adecuado para la gestión sistemas críticos utilizando herramientas de carga de alta velocidad y otros mecanismos de mejora del rendimiento.

3. **Alta disponibilidad**: solidez y disponibilidad constante son características distintivas de MySQL. Ofrece una amplia variedad de soluciones de alta disponibilidad, desde replicación a servidores de cluster especializados, u ofertas de terceros.

4. **Robusto soporte transaccional**: MySQL ofrece uno de los motores de bases de datos transaccionales más potentes del mercado. Las características incluyen un soporte completo de ACID (atómica, consistente, aislada, duradera), bloqueo a nivel de filas, posibilidad de transacciones distribuidas, y soporte de transacciones con múltiples versiones donde los lectores no bloquean a los escritores y viceversa. También se asegura una integridad completa de los datos mediante integridad referencial, niveles de aislamiento de transacciones especializados, y detección de *deadlocks*.

5. **Fortalezas en Web**: MySQL es el estÁndar para sitios web de gran tráfico por su motor de consultas de alto rendimiento, su posibilidad de insertar datos a gran velocidad, y un buen soporte para funciones web especializadas como las búsquedas *fulltext*. Otras características como las tablas en memoria y tablas comprimidas hasta un 80 % hacen de MySQL una buena opción para aplicaciones web.

6. **Fuerte protección de datos**: MySQL ofrece características de seguridad que aseguran una protección absoluta de los datos. En cuanto a autenticación, MySQL ofrece potentes mecanismos para asegurar que sólo los usuarios autorizados tienen acceso al servidor. También se ofrece soporte SSH y SSL para asegurar conexiones seguras. Existe una estructura de privilegios que permite que los usuarios sólo puedan acceder a los datos que se les permite, así como potentes funciones de cifrado y descifrado para asegurarse de que los datos están protegidos. Finalmente, se ofrecen utilidades de *backup* y recuperación por parte de MySQL y terceros, que permiten copias completas, tanto lógicas como físicas.

7. **Desarrollo completo de aplicaciones**: Uno de los motivos por los que MySQL es la base de datos *open source* más popular es que ofrece un soporte completo

para cualquier necesidad de desarrollo. En la base de datos se puede encontrar soporte para procedimientos almacenados, *triggers*, funciones, vistas, cursores, SQL estándar, y mucho más. Existen librerías para dar soporte a MySQL en aplicaciones empotradas. También se ofrecen *drivers* (ODBC, JDBC, ...) que permiten que distintos tipos de aplicaciones puedan usar MySQL como gestor de bases de datos. No importa si es PHP, Perl, Java, Visual Basic, o .NET, MySQL ofrece a los desarrolladores todo lo que necesitan para conseguir el éxito en el desarrollo de sistemas de información basados en bases de datos.

8. **Facilidades de gestión**: MySQL ofrece posibilidades de instalación excepcionales, con un tiempo medio desde la descarga hasta completar la instalación de menos de quince minutos. Esto es cierto sin importar que la plataforma sea Windows, Linux, Macintosh, o UNIX. Una vez instalado, características de gestión automáticas como expansión automática del espacio, o los cambios dinámicos de configuración descargan parte del trabajo de los administradores. MySQL también ofrece una completa colección de herramientas gráficas de gestión que permiten al administrador gestionar, controlar y resolver problemas en varios servidores desde una misma estación de trabajo. Además, hay multitud de herramientas de terceros que gestionan tareas como el diseño de datos, administración, gestión de tareas y monitorización.

9. **Open source y soporte 24/7**: Muchas empresas no se atreven a adoptar *software open source* porque creen que no podrán encontrar el tipo de soporte o servicios profesionales en los que confían con su *software* propietario. Las preguntas sobre indemnizaciones también aparecen. Estas preocupaciones pueden desaparecer con el completo servicio de soporte e indemnización disponibles. MySQL no es un proyecto típico *Open Source* ya que cuenta con un modelo de coste y soporte que ofrece una combinación única entre la libertad del código abierto y la confianza de un *software* con soporte.

10. **Coste total de propiedad menor**: al migrar aplicaciones actuales a MySQL, o usar MySQL para nuevos desarrollos, las empresas estan ahorrando costes que muchas veces llegan a las siete cifras. Las empresas están descubriendo que pueden alcanzar niveles sorprendentes de escalabilidad y rendimiento, y todo a un coste bastante menor que el de los sistemas propietarios. Además, la robustez y facilidad de mantenimiento de MySQL implican que los administradores no pierden el tiempo con problemas de rendimiento o disponibilidad, sino que pueden concentrarse en tareas de mayor impacto en el negocio.

3.2.3. MySQL vs PostgreSQL

MySQL y PostgreSQL son, sin duda, las dos bases de datos libres más empleadas. En general, cuando se hacen comparativas entre estas dos herramientas, se

tiende a destacar las ventajas y desventajas que históricamente ha tenido cada una de ellas, sin tener en cuenta la gran evolución producida en ambos gestores de bases de datos.

Cierto es que, en MySQL el principal objetivo de diseño fue la velocidad de forma que se sacrificaron algunas características esenciales que otros sistemas ofrecían. Sin embargo, MySQL ha recorrido un largo camino en la adición de funcionalidades avanzadas. Se habla por ejemplo de la falta de integridad referencial en MySQL, pero esta afirmación es únicamente válida para versiones anteriores a la 4.0. Por otra parte, siempre se habla de la lentitud de PostgreSQL frente a MySQL, sin tener en cuenta que que PostgreSQL ha mejorado enormemente su velocidad en los últimos lanzamientos.

En general, se pueden sacar las siguientes conclusiones sobre las comparativas entre las dos herramientas:

- **MySQL**:
  - Su principal objetivo de diseño fue la velocidad.
  - Consume muy pocos recursos, tanto de CPU como de memoria.
  - Licencia GPL a partir de la versión 3.23.19.
  - Mayor rendimiento. Mayor velocidad tanto al conectar con el servidor como al servir consultas.
  - Mejores utilidades de administración (backup, recuperación de errores, etc).
  - Mejor integración con PHP.
  - No hay límites en el tamaño de los registros.
  - Mejor control de acceso.
  - Mayor facilidad de administración.

  Los inconvenientes de MySQL son:
  - Ofrece menor garantía de integridad en los datos.
  - Licencia más restrictiva.
  - Se comporta peor en entornos de alta carga.
- **PostgreSQL**:
  - PostgreSQL intenta ser un sistena de bases de datos de mayor nivel que MySQL, a la altura de Oracle, Sybase o Interbase.
  - Licencia BSD.

- Por su arquitectura de diseño, escala muy bien al aumentar el número de CPUs y la cantidad de RAM.
- Tiene mejor soporte para triggers y procedimientos en el servidor.
- Ofrece una garantía de integridad en los datos mucho más fuerte que MySQL. En aquellos escenarios en los cuales no podemos permitirnos que se corrompa o se pierda ni un solo registro (por ejemplo, aplicaciones médicas o bancarias) es más recomendable usar esta opción.
- Aunque sea más lenta respondiendo a una única consulta, PostgreSQL presenta una mejor escalabilidad y rendimiento bajo grandes cargas de trabajo.

Como inconvenientes de PostgreSQL, se pueden citar:

- Consume bastantes más recursos y carga más el sistema.
- Límite del tamaño de cada fila de las tablas a 8k, aunque se puede ampliar a 32k recompilando, pero con un coste añadido en el rendimiento.
- Es más lenta que MySQL.
- Tiene menos funciones en PHP.

Como conclusión sobre la comparativa, puede decirse que ninguno de estos dos gestores son totalmente perfectos, por lo que no hay que obcecarse en una elección única y fanática de alguno de ellos. Simplemente se trata de escoger el más conveniente en cada caso. La decisión sobre qué herramienta elegir podría trasladarse a una decisión de velocidad frente a potencia.

Teniendo en cuenta las dimensiones de este proyecto, las cuales no son grandes, que la carga de trabajo no es alta y la seguridad no es un punto crítico, MySQL es un gestor que se adecúa a las necesidades del mismo.

3.3. JDBC

Una de las desventajas principales de las primeras versiones de Java era que no tenía soporte alguno para el acceso a bases de datos, lo cual limitó la utilidad de Java en el campo de los negocios. Sin embargo, a partir de la versión 1.1 del JDK, Java proporciona un soporte completo para bases de datos por medio de JDBC (Java Database Connectivity). JDBC es un API compuesto por un conjunto de clases y métodos de operación que permiten a cualquier programa Java acceder a sistemas de bases de datos de forma homogénea. JDBC permite establecer una conexión con una base de datos, enviar sentencias SQL y procesar los resultados.

3.3.1. Drivers

Todos los programas Java que se conectan a una base de datos vía JDBC, utilizan un *driver* o controlador de bases de datos. Es el intermediario entre la capa de negocio y la capa de base de datos y así mismo actúa como el traductor de las sentencias Java a sentencias SQL propias del manejador de bases de datos.

Los drivers JDBC pueden clasificarse en una de las siguientes cuatro categorías [34]:

1. **JDBC-ODBC bridge**: La primera categoría de drivers es la utilizada por Sun inicialmente para popularizar JDBC y consiste en aprovechar todo lo existente, estableciendo un puente entre JDBC y ODBC. Este driver convierte todas las llamadas JDBC a llamadas ODBC y realiza la conversión correspondiente de los resultados. La ventaja de este controlador proporcionado con el JDK, es que Java dispone de acceso inmediato a todas las fuentes posibles de bases de datos y no hay que hacer ninguna configuración adicional aparte de la ya existente. No obstante, tiene dos desventajas muy importantes. Por un lado, la mayoría de los *drivers* ODBC a su vez convierten sus llamadas a llamadas a una librería nativa del fabricante DBMS, con lo cual la lentitud del driver JDBC-ODBC puede ser exasperante, al llevar dos capas adicionales que no añaden funcionalidad alguna. Por otra parte, el puente JDBC-ODBC requiere una instalación ODBC ya existente y configurada.

   Lo anterior implica que para distribuir con seguridad una aplicación Java que use JDBC habría que limitarse en primer lugar a entornos Windows (donde está definido ODBC) y en segundo lugar, proporcionar los controladores ODBC adecuados y configurarlos correctamente. Esto hace que este tipo de *drivers* esté totalmente descartado en el caso de aplicaciones comerciales, e incluso en cualquier otro desarrollo, debe ser considerado como una solución transitoria, porque el desarrollo de drivers totalmente en Java hará innecesario el uso de estos puentes.

2. **Java/Binario**: Este *driver* se salta la capa ODBC y habla directamente con la libreráa nativa del fabricante del sistema DBMS. Este driver es un driver 100 % Java pero aún así necesita la existencia de un código binario (la librería DBMS) en la máquina del cliente, con las limitaciones y problemas que esto implica.

3. **100 % Java/Protocolo nativo**: Es un *driver* realizado completamente en Java que se comunica con el servidor DBMS utilizando el protocolo de red nativo del servidor. De esta forma, el driver no necesita intermediarios para hablar con el servidor y convierte todas las peticiones JDBC en peticiones de red contra el servidor. La ventaja de este tipo de driver es que es una solución 100 % Java y, por lo tanto, independiente de la máquina en la que se va a ejecutar el programa.

Igualmente, dependiendo de la forma en que esté programado, puede no necesitar ninguna clase de configuración por parte del usuario. La única desventaja de este tipo de controladores es que el cliente está ligado a un servidor DBMS concreto, ya que el protocolo de red que utiliza MS SQL Server por ejemplo no tiene nada que ver con el utilizado por DB2, MySQL u Oracle. La mayoría de los fabricantes de bases de datos han incorporado a sus propios *drivers* JDBC del segundo o tercer tipo, con la ventaja de que no suponen un coste adicional.

4. **100 % Java/Protocolo independiente**: Esta es la opción más flexible ya que se trata de un driver 100 % Java que requiere la presencia de un intermediario en el servidor. En este caso, el driver JDBC hace las peticiones de datos al intermediario en un protocolo de red independiente del servidor DBMS. El intermediario a su vez, que está ubicado en el lado del servidor, convierte las peticiones JDBC en peticiones nativas del sistema DBMS. La ventaja de este método es inmediata: el programa que se ejecuta en el cliente, y aparte de las ventajas de los drivers 100 % Java, también presenta la independencia respecto al sistema de bases de datos que se encuentra en el servidor.

   De esta forma, si una empresa distribuye una aplicación Java para que sus usuarios puedan acceder a su servidor MS SQL y posteriormente decide cambiar el servidor por Oracle, MySQL o DB2, no necesita volver a distribuir la aplicación, sino que únicamente debe reconfigurar la aplicación residente en el servidor que se encarga de transformar las peticiones de red en peticiones nativas. La única desventaja de este tipo de *drivers* es que la aplicación intermediaria es una aplicación independiente que suele tener un coste adicional por servidor físico, que hay que añadir al coste del servidor de bases de datos.

3.3.2. Clases

JDBC ofrece el paquete java.sqlen el que existen clases muy útiles para trabajar con bases de datos:

- *DriverManager*: Esta clase se utiliza para cargar el controlador.
- *Connection*: Clase usada para establecer conexiones con las bases de datos.
- *Statement*: Con esta clase se pueden crear consultas SQL y enviarlas a la base de datos.
- *ResultSet*: En esta clase se almacena el resultado de la consulta.

3.3.3. JDBC frente vs ODBC

El ODBC de Microsoft (Open Database Connectivity), es probablemente el API más extendido para el acceso a bases de datos relacionales. Ofrece la posibilidad de conectar a la mayoría de las bases de datos en casi todas las plataformas. Se puede usar ODBC desde Java, pero es preferible hacerlo con la ayuda de JDBC mediante el puente JDBC-ODBC. La necesidad de usar JDBC viene de [33]:

- ODBC no es apropiado para su uso directo con Java porque usa una interface C. Las llamadas desde Java a código nativo C tienen un número de inconvenientes en la seguridad, la implementación, la robustez y en la portabilidad automática de las aplicaciones.
- Una traducción literal del API C de ODBC en el API Java podría no ser deseable. Por ejemplo, Java no tiene punteros, y ODBC hace un uso copioso de ellos. Se puede pensar en JDBC como un ODBC traducido a una interfase orientada a objeto que es el natural para programadores Java.
- ODBC es difícil de aprender. Mezcla caracter'ssticas simples y avanzadas juntas, y sus opciones son complejas para sentencias simples. JDBC por otro lado, ha sido diseñado para mantener la sencillez mientras que permite el uso de las características avanzadas cuando éstas son necesarias.
- Un API Java como JDBC es necesario en orden a permitir una solución Java pura. Cuando se usa ODBC, el gestor de *drivers* de ODBC y los *drivers* deben instalarse manualmente en cada máquina cliente. Como el *driver* JDBC esta completamente escrito en Java, el código JDBC es automáticamente instalable, portable y seguro en todas las plataformas Java.

En resumen, el API JDBC es el interfaz natural de Java para las abstracciones y conceptos básicos de SQL. Además, retiene las características básicas de diseño de ODBC.

Capítulo 4

HERRAMIENTA TMM

4.1. Análisis y diseño

4.1.1. Introducción

Se pretende construir una herramienta que permita movilizar los procesos de negocio asociados a la actividad de una empresa de transporte de mercancías.

Nomenclatura

- **Entrega:** Expedición con origen la central y destino el destinatario.
- **Recogida:** Expedición con origen el cliente y destino la central.
- **Notificación:** Comunicado interno a los transportistas.
- **Hoja de ruta:** Conjunto de entregas y recogidas asignadas a un conductor.

Algunas características generales de la herramienta son:

- Es escalable de forma que puede utilizarse por empresas de distinto tamaño en las que parámetros como el número de transportistas o el volumen de datos manejados pueden variar. La adaptación de la herramienta a una determinada empresa se haría tras un estudio de dimensionamiento en el que se determinará qué características deberían cumplir los equipos de hardware para dar servicio en unas condiciones específicas.

- Es independiente del medio de transporte empleado para la actividad: puede ser usada por transportistas con medios motorizados, medios no motorizados o incluso para repartos/recogidas a pie.

- Se asume que la asignación de las expediciones a cada transportista se realiza siempre desde la central y no entre conductores.

- El módulo cliente móvil de la herramienta puede funcionar en modo *offline* si se producen pérdidas de connectividad en el dispositivo. De esta forma el transportista no se verá obligado a interrumpir su actividad por un problema de conexión.

- Los usuarios finales no requieren un dispositivo específico asignado, sino que pueden usar la aplicación en cualquier dispositivo en el que esté instalada.

Como se ha comentado en anteriores capítulos, la herramienta estará formada por varios módulos: un cliente móvil para Android, dos clientes web para perfiles de oficina, web services y base de datos.

En cuanto al módulo Android, la aplicación necesita conexión a internet bien mediante 3G o mediante Wi-fi. Esta conexión es necesaria sólo para algunas de las operaciones que se realizan en el dispositivo, las cuales son:

1. Validación de usuario: El conductor deberá validarse en la aplicación facilitando su número de usuario y su contraseña, datos que son enviados al servidor para su análisis. La respuesta del servidor en esta conexión determinará si el usuario puede hacer uso de la aplicación o no puede acceder a ella.

2. Recepción de expediciones: En la primera sincronización el usuario recibirá del servidor todas las expediciones que le hayan sido asignadas en su hoja de ruta. Si esta conexión falla, el usuario no visualizará ningún dato en el terminal, aunque tiene la opción de activar una sincronización en cualquier momento a través de uno de los items de un menú.

3. Visualización de mapas: La aplicación hace uso del servicio Google Maps para la visualización de mapas, por lo que la conexión a internet es imprescindible para esta comunicación.

4. Cierre de la aplicación: El dispositivo realiza siempre una conexión con el servidor para enviarle las últimas modificaciones de sus datos antes de cerrar la aplicación.

Exceptuando los casos anteriormente mencionados, el cliente Android puede funcionar sin conexión a internet para el resto de las funcionalidades que ofrece, las cuales serán comentadas en detalle posteriormente.

La aplicación hace uso también del componente GPS integrado en el dispositivo. Esto permite al usuario que porta el terminal visualizar su localización en los mapas de la herramienta y al jefe de flota visualizar la ruta seguida por dicho usuario. Si el GPS no está disponible, la aplicación continúa con un correcto funcionamiento, pero ninguno de los perfiles mencionados podrá visualizar datos relativos a la posición del transportista.

En el caso de los clientes web para los equipos de oficina, la conexión a internet es necesaria en todo momento tal y como es evidente.

Previamente a entrar en detalle en las funcionalidades que ofrece la herramienta, se expondrá la arquitectura que sigue el sistema para proporcionar una visión más clara de todo el conjunto y de la conexión de los distintos módulos.

4.1.2. Arquitectura

En la figura 4.1 se muestra un esquema de la arquitectura de la herramienta, el cual se explica a continuación:

- **Acceso de usuarios con perfil móvil:** Los conductores accederán desde la aplicación instalada en sus terminales Android. Ésta se comunicará con el servidor mediante servicios web para recibir y enviar expediciones y notificaciones, así como reportar su posición cada cierto período de tiempo.
- **Acceso de usuarios con perfil web:** El personal en la central accederá al sistema a través del navegador web de su equipo, el cual comunica con su correspondiente aplicación residente en el servidor web.
- **Servidor web:** Este servidor alojará tanto los web services que comunican con el cliente Android como los servlets pertenecientes a las aplicaciones de los perfiles web. Hará de pasarela entre los clientes finales y la base de datos. Para el desarrollo de este proyecto, el servidor usado tiene los siguientes requerimientos instalados:
  - JDK 1.6.0_11
  - Servidor Apache Tomcat 6.0.18
  - BBDD MySQL Server 5.1
  - Conector MySQL Java 5.1.7

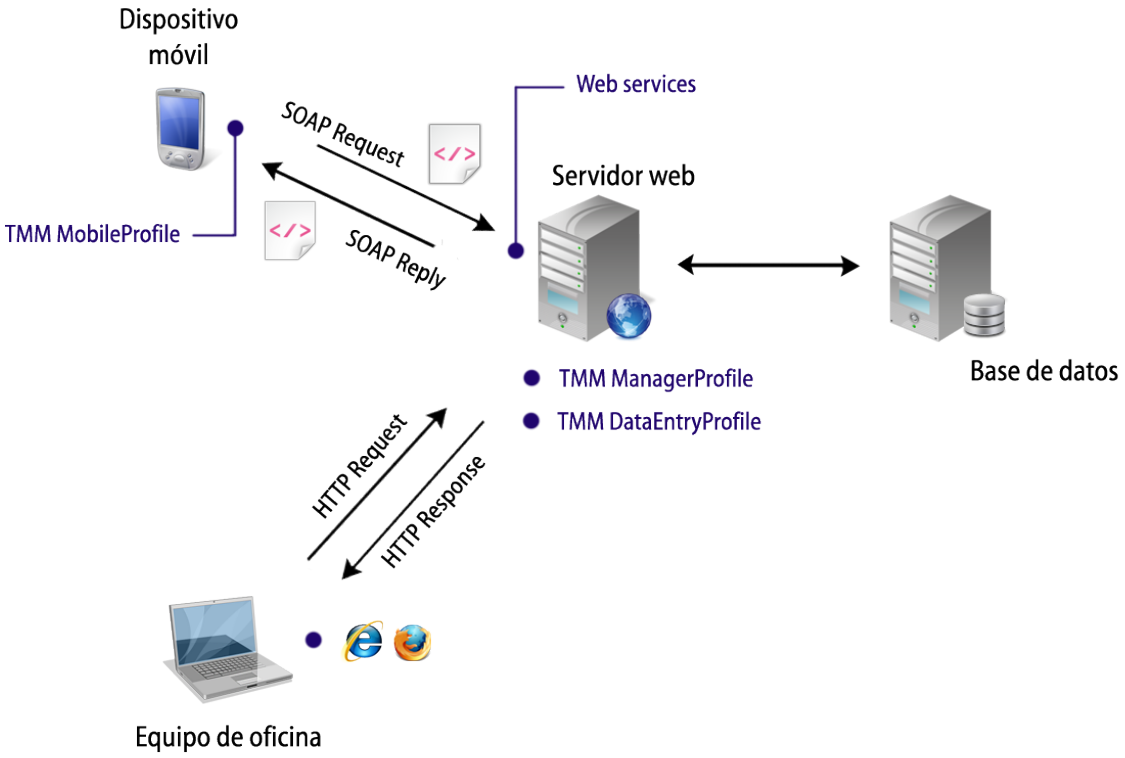

Figura 4.1: Diagrama de la arquitectura de la herramienta TMM.

- **Servidor BD:** En este diagrama se ha representado la base de datos alojada en un servidor distinto del servidor web para mayor claridad conceptual, pero dicha base de datos podría estar alojada también en el servidor web. En dicha BD se almacenarán los datos de los usuarios del sistema así como de las expediciones y notificaciones a gestionar.

La comunicación entre el cliente Android y el servidor se realiza usando el protocolo SOAP, el cual hace una comunicación bajo HTTP basada en el intercambio de información a través de documentos XML, como ya se ha explicado en capítulos anteriores. Cabe destacar que, gracias a su uso a través de HTTP, evita la restricción de los *firewalls*.

4.1.3. Casos de uso

En esta sección se explicará más en detalle las funcionalidades que ofrece la herramienta para generar en el lector una visión clara de las posibilidades que ofrece este sistema y de su funcionamiento.

Los usuarios de *TMM* se pueden clasificar según tres tipos: *Manager*, *Transportista* y *DataEntry*. Estos son los tipos que están definidos en la base de datos, de forma que sólo determinados tipos de usuarios pueden acceder a determinados módulos de la herramienta, según corresponda.

Cliente Android:

Las operaciones que puede realizar un transportista desde su terminal son las siguientes:

- Iniciar sesión en la aplicación. Esta funcionalidad constituye una medida de seguridad, de forma que únicamente podrán hacer uso de la herramienta personas autorizadas para ello.
- Visualizar listados de entregas, recogidas y notificaciones. El usuario puede ver rápidamente el número de expediciones y notificaciones y los datos más relevantes de ellas.
- Visualizar detalles de cada expedición. El usuario puede conocer información más detallada de cada entrega y recogida, como por ejemplo los clientes, direcciones, medidas de los paquetes, etc.
- Leer y borrar de la aplicación notificaciones enviadas desde la central.

- Realizar reportes al servidor. El usuario puede reportar como realizadas, no realizadas o rechazadas sus expediciones.
- Sincronizar datos con el sistema en cualquier momento. La aplicación incluye un botón para sincronizar instantáneamente los datos almacenados en el dispositivo móvil.
- Cambiar idioma. La aplicación está disponible en español y en inglés. El usuario puede seleccionar uno de estos dos idiomas en cualquier momento.
- Acceso directo a otras aplicaciones útiles. En la herramienta TMM se han integrado dos aplicaciones ya existentes que pueden ser de gran utilidad al transportista. Dichas aplicaciones son *Places Directory*, desarrollada por Google y *Gasolineras España*, desarrollada por la empresa española Mobialia [35].
- Visualización de expediciones en mapas. El conductor puede ver sus expediciones posicionadas en mapas con diferentes vistas. Cada expedición se representa en el mapa con el color que identifica su estado. Además, al tocar sobre cada una de ellas aparece información más detallada de la misma.
- Seleccionar tipo de expediciones a visualizar en el mapa. La aplicación ofrece varias posibilidades para visualizar las expediciones: mostrar sólo entregas, mostrar sólo recogias o mostrar ambas. Además, el usuario puede elegir que se muestren únicamente las expediciones que se encuentren en un estado determinado.
- Visualización de la ubicación propia en mapas. Con esta funcionalidad, el usuario puede ver en mapas con diferentes vistas cuál es su posición exacta.
- Manejo de mapas. El usuario puede elegir entre las vistas mapa, satélite y *Street View*, y puede seleccionar ver los mapas en pantalla normal o pantalla completa. Además es posible hacer zoom sobre el mapa así como desplazarlo hacia la posición interesada.
- Salir de la aplicación. La aplicación no se cerrará completamente hasta que el usuario así lo indique a través del interfaz gráfico.

Además de estas funciones realizadas por el usuario, la propia herramienta realiza también otras funciones que no requieren de la actuación del usuario para su inicio. Son las siguientes:

- Notificación de nuevas recogidas recibidas. Cuando la aplicación recibe recogidas nuevas desde el servidor, ésta notifica al usuario mediante vibración, alarma sonora y encendido del LED. Además se mostrará la notificación en la barra de estado del mismo modo que se muestran las notificaciones estándar de Android: con un icono

identificativo de la aplicación y un mensaje en la ventana de notificaciones, a través de la cual el usuario tendrá acceso directo a la pantalla de listado de recogidas. La notificación se eliminará tras ser consultada por el usuario.

- Sincronizaciones en *background*. Esta tarea se describirá con detalle más adelante, pero básicamente consiste en la sincronización con el servidor y actualización de los datos del dispostivo sin mediación del usuario.

Perfil web de gestión:

Las operaciones que el usuario puede realizar desde su navegador son las siguientes:

- Inicio de sesión en la aplicación. El usuario deberá facilitar su número de usuario y contraseña para acceder a la aplicación. Sólo a los usuarios de tipo *Manager* se les permitirá el acceso.
- Gestión de usuarios. Será posible el alta, modificación y eliminación de usuarios con perfiles *Manager* y *Transportista*.
- Seguimiento de la actividad de un determinado transportista. El usuario podrá visualizar en mapas la ruta seguida por cualquier transportista que seleccione desde el listado. En este mismo mapa se muestran también las expediciones que el transportista ha reportado, representadas por el color que identifica el estado en que se encuentran.
- Seguimiento de entregas. En el interfaz se muestran todas las entregas existentes en el sistema en forma de listado con los campos más importantes. El usuario podrá acceder a información más detallada de una entrega al seleccionarla.
- Seguimiento de recogidas. En el interfaz se muestran todas las recogidas existentes en el sistema en forma de listado con los campos más importantes. El usuario podrá acceder a información más detallada de la recogida al seleccionarla.
- Gestión de notificaciones. Será posible la creación, modificación y eliminación de notificaciones.
- Uso de filtros para facilitar la búsqueda. El usuario podrá hacer uso de filtros para buscar la información deseada relativa a los usuarios, entregas, recogidas y notificaciones.

Perfil web para introducción de expediciones:

Como ya se ha explicado anteriormente, este módulo se ha desarrollado con el fin de poder introducir en el sistema expediciones desde un interfaz web, de forma que el proyecto quede más completo.

- Inicio de sesión. Para mantener la línea de diseño que siguen los módulos comentados anteriormente, será necesario iniciar sesión para poder usar la herramienta. El perfil de este usuario se ha denominado en la base de datos *DataEntry*, y deberá introducir su número de usuario y contraseña para poder realizar operaciones.
- Creación de entregas. El usuario dispone de un formulario en el que rellenar los campos característicos de una entrega.
- Creación de recogidas. El usuario dispone de un formulario en el que rellenar los campos característicos de una recogida.

Servicios web:

Aparte de la función principal que tienen los servicios web de intermediar entre la base de datos y el cliente Android, se ha implementado un sistema de *logs* en los métodos invocados por el cliente. En estos ficheros se imprime información sobre qué datos se han enviado y recibido en las comunicaciones, así como la información más relevante sobre los mismos. Estos *logs* son muy valiosos para rastreo de incidencias o errores.

4.1.4. Modelo de datos

Para proceder a la creación de la base de datos, primero se ha estudiado el universo de discurso para generar el diagrama entidad-relación del modelo: seleccionando los conjuntos entidad, describiendo las propiedades de cada conjunto entidad y seleccionando los conjuntos asociación.

En la figura 4.2 se muestra el diagrama ER a partir del cual se construirán las sentencias SQL necesarias para la generación de la base de datos.

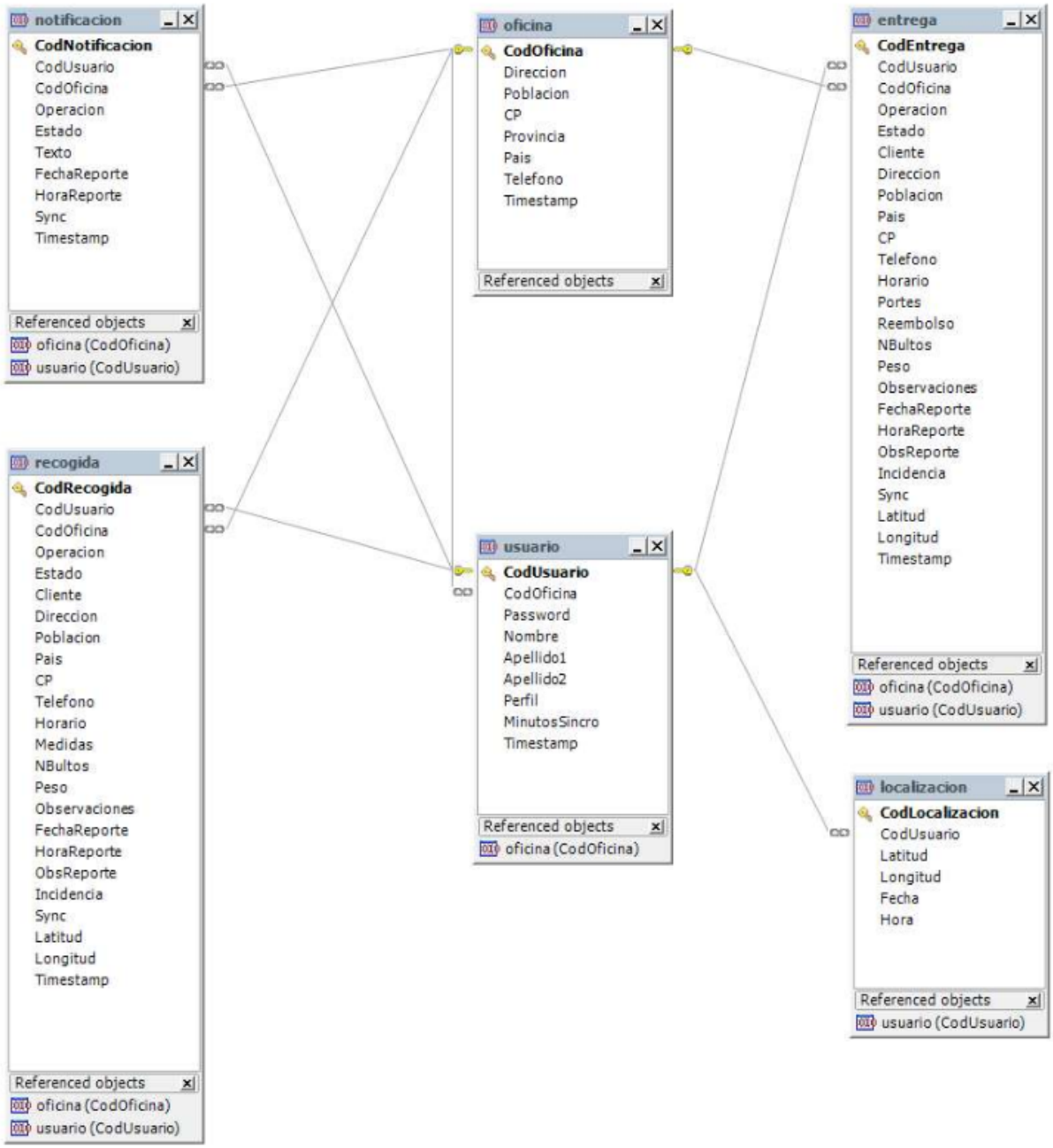

Figura 4.2: [Modelo de datos] Diagrama Entidad-Relación para la herramienta TMM.

La sentencia SQL que crea la base de datos es la siguiente:

```
CREATE DATABASE tmm
DEFAULT CHARACTER SET utf8
COLLATE utf8_spanish_ci;
```

Código 4.1: [Modelo de datos] Sentencia SQL de creación de la base de datos.

Dicha base de datos consta de las siguientes seis tablas:

- **Tabla Oficina:** Esta tabla almacena los datos relacionados con la situación y el contacto de cada oficina. Esta tabla ha sido creada para que empresas de gran tamaño con varias oficinas puedan hacer uso del sistema. El identificador único de la tabla es el campo *CodOficina*.

```
CREATE TABLE `oficina` (
    `CodOficina` varchar(5) COLLATE utf8_spanish_ci NOT NULL,
    `Direccion` varchar(50) COLLATE utf8_spanish_ci NOT NULL,
    `Poblacion` varchar(20) COLLATE utf8_spanish_ci NOT NULL,
    `CP` int(8) NOT NULL,
    `Provincia` varchar(20) COLLATE utf8_spanish_ci DEFAULT NULL,
    `Pais` varchar(20) COLLATE utf8_spanish_ci NOT NULL,
    `Telefono` varchar(15) COLLATE utf8_spanish_ci DEFAULT NULL,
    `Timestamp` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (`CodOficina`)
)   ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;
```

Código 4.2: [Modelo de datos] Sentencia SQL de creación de la tabla *Oficina*.

- **Tabla Usuario:** Esta tabla almacena los datos de todos los usuarios del sistema. Posee una clave primaria y una foránea, constituidas por los campos *CodUsuario* y *CodOficina*, respectivamente. La tabla contiene información necesaria para el inicio de sesión de cada usuario en la herramienta, es decir, los campos *CodUsuario*, *Password* y *Perfil*. Además cabe destacar el campo *MinutosSincro*, gracias al cual puede configurarse el período entre sincronizaciones desde el cliente móvil de forma individual para cada transportista.

```
CREATE TABLE `usuario` (
       `CodUsuario` varchar(5) COLLATE utf8_spanish_ci NOT NULL,
```

```
        `CodOficina` varchar(5) COLLATE utf8_spanish_ci NOT NULL,
        `Password` varchar(5) COLLATE utf8_spanish_ci NOT NULL,
        `Nombre` varchar(20) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Apellido1` varchar(20) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Apellido2` varchar(20) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Perfil` varchar(20) COLLATE utf8_spanish_ci NOT NULL,
        `MinutosSincro` int(2) NOT NULL DEFAULT '15',
        `Timestamp` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
            ON UPDATE CURRENT_TIMESTAMP,
        PRIMARY KEY (`CodUsuario`),
        UNIQUE KEY `UQ_USUARIO` (`Nombre`,`Apellido1`,`Apellido2`),
        KEY `FK_USUARIO_OFICINA` (`CodOficina`),
        CONSTRAINT `FK_USUARIO_OFICINA` FOREIGN KEY (`CodOficina`) REFERENCES
            `oficina` (`CodOficina`) ON DELETE CASCADE ON UPDATE CASCADE
    )   ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;
```

Código 4.3: [Modelo de datos] Sentencia SQL de creación de la tabla *Usuario*.

- **Tabla Entrega:** Cada registro de esta tabla almacena los datos relativos a una expedición de tipo entrega. Su identificador único está formado por la clave primaria *CodEntrega* y las claves foráneas *CodUsuario* y *CodOficina*. Almacena los campos usuales para una entidad de este tipo como son la dirección de entrega o el cliente. También contiene campos necesarios para el funcionamiento de la herramienta, como por ejemplo campos para información de reporte o campos necesarios para el posicionamiento de la entrega en mapas. Los campos de reporte llegarán vacíos desde el servidor y serán rellenados en el momento en que el transportista reporte la entrega en cuestión.

```
CREATE TABLE `entrega` (
        `CodEntrega` int(8) NOT NULL AUTO_INCREMENT,
        `CodUsuario` varchar(5) COLLATE utf8_spanish_ci NOT NULL,
        `CodOficina` varchar(5) COLLATE utf8_spanish_ci NOT NULL,
        `Operacion` varchar(1) COLLATE utf8_spanish_ci NOT NULL,
        `Estado` int(1) NOT NULL,
        `Cliente` varchar(50) COLLATE utf8_spanish_ci NOT NULL,
        `Direccion` varchar(50) COLLATE utf8_spanish_ci NOT NULL,
        `Poblacion` varchar(30) COLLATE utf8_spanish_ci NOT NULL,
        `Pais` varchar(20) COLLATE utf8_spanish_ci DEFAULT NULL,
        `CP` int(8) NOT NULL,
        `Telefono` varchar(15) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Horario` varchar(40) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Portes` decimal(7,2) NOT NULL DEFAULT '0.00',
        `Reembolso` decimal(7,2) NOT NULL DEFAULT '0.00',
```

```
        `NBultos` int(2) DEFAULT NULL,
        `Peso` decimal(4,3) NOT NULL DEFAULT '0.000',
        `Observaciones` varchar(50) COLLATE utf8_spanish_ci DEFAULT NULL,
        `FechaReporte` varchar(8) COLLATE utf8_spanish_ci DEFAULT NULL,
        `HoraReporte` varchar(5) COLLATE utf8_spanish_ci DEFAULT NULL,
        `ObsReporte` varchar(150) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Incidencia` varchar(20) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Sync` int(1) NOT NULL,
        `Latitud` varchar(40) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Longitud` varchar(40) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Timestamp` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
            ON UPDATE CURRENT_TIMESTAMP,
        PRIMARY KEY (`CodEntrega`),
        KEY `FK_ENTREGA_OFICINA` (`CodOficina`),
        KEY `FK_ENTREGA_USUARIO` (`CodUsuario`),
        CONSTRAINT `FK_ENTREGA_OFICINA` FOREIGN KEY (`CodOficina`) REFERENCES
            `oficina` (`CodOficina`) ON DELETE CASCADE ON UPDATE CASCADE,
        CONSTRAINT `FK_ENTREGA_USUARIO` FOREIGN KEY (`CodUsuario`) REFERENCES
            `usuario` (`CodUsuario`) ON DELETE CASCADE ON UPDATE CASCADE
    )   ENGINE=InnoDB AUTO_INCREMENT=20 DEFAULT CHARSET=utf8
            COLLATE=utf8_spanish_ci;
```

Código 4.4: [Modelo de datos] Sentencia SQL de creación de la tabla *Entrega*.

- **Tabla Recogida:** Cada registro de esta tabla almacena los datos relativos a una expedición de tipo recogida. Su identificador único está formado por la clave primaria *CodRecogida* y las claves foráneas *CodUsuario* y *CodOficina*. Almacena los campos usuales para una entidad de este tipo y campos necesarios para el sistema, como por ejemplo campos para información de reporte o campos necesarios para el posicionamiento de la recogida en mapas.

```
CREATE TABLE `recogida` (
        `CodRecogida` int(8) NOT NULL AUTO_INCREMENT,
        `CodUsuario` varchar(5) COLLATE utf8_spanish_ci NOT NULL,
        `CodOficina` varchar(5) COLLATE utf8_spanish_ci NOT NULL,
        `Operacion` varchar(1) COLLATE utf8_spanish_ci NOT NULL,
        `Estado` int(1) NOT NULL,
        `Cliente` varchar(50) COLLATE utf8_spanish_ci NOT NULL,
        `Direccion` varchar(50) COLLATE utf8_spanish_ci NOT NULL,
        `Poblacion` varchar(30) COLLATE utf8_spanish_ci NOT NULL,
        `Pais` varchar(20) COLLATE utf8_spanish_ci DEFAULT NULL,
        `CP` int(8) NOT NULL,
        `Telefono` varchar(15) COLLATE utf8_spanish_ci DEFAULT NULL,
```

```
        `Horario` varchar(40) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Medidas` varchar(50) COLLATE utf8_spanish_ci DEFAULT NULL,
        `NBultos` int(2) DEFAULT NULL,
        `Peso` decimal(4,3) NOT NULL DEFAULT '0.000',
        `Observaciones` varchar(50) COLLATE utf8_spanish_ci DEFAULT NULL,
        `FechaReporte` varchar(8) COLLATE utf8_spanish_ci DEFAULT NULL,
        `HoraReporte` varchar(5) COLLATE utf8_spanish_ci DEFAULT NULL,
        `ObsReporte` varchar(150) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Incidencia` varchar(20) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Sync` int(1) NOT NULL,
        `Latitud` varchar(40) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Longitud` varchar(40) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Timestamp` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
            ON UPDATE CURRENT_TIMESTAMP,
        PRIMARY KEY (`CodRecogida`),
        KEY `FK_RECOGIDA_OFICINA` (`CodOficina`),
        KEY `FK_RECOGIDA_USUARIO` (`CodUsuario`),
        CONSTRAINT `FK_RECOGIDA_OFICINA` FOREIGN KEY (`CodOficina`) REFERENCES
            `oficina` (`CodOficina`) ON DELETE CASCADE ON UPDATE CASCADE,
        CONSTRAINT `FK_RECOGIDA_USUARIO` FOREIGN KEY (`CodUsuario`) REFERENCES
            `usuario` (`CodUsuario`) ON DELETE CASCADE ON UPDATE CASCADE
    )   ENGINE=InnoDB AUTO_INCREMENT=12 DEFAULT CHARSET=utf8
            COLLATE=utf8_spanish_ci;
```

Código 4.5: [Modelo de datos] Sentencia SQL de creación de la tabla *Recogida*.

- **Tabla Notificación:** Cada registro de esta tabla almacena los datos relativos a una notificación. Su identificador único está formado por la clave primaria *CodNotificación* y las claves foráneas *CodUsuario* y *CodOficina*. Almacena tanto el texto del mensaje como campos para información de reporte o estado.

```
CREATE TABLE `notificacion` (
        `CodNotificacion` int(11) NOT NULL AUTO_INCREMENT,
        `CodUsuario` varchar(5) COLLATE utf8_spanish_ci NOT NULL,
        `CodOficina` varchar(5) COLLATE utf8_spanish_ci NOT NULL,
        `Operacion` varchar(1) COLLATE utf8_spanish_ci NOT NULL,
        `Estado` int(1) NOT NULL,
        `Texto` varchar(160) COLLATE utf8_spanish_ci DEFAULT NULL,
        `FechaReporte` varchar(8) COLLATE utf8_spanish_ci DEFAULT NULL,
        `HoraReporte` varchar(5) COLLATE utf8_spanish_ci DEFAULT NULL,
        `Sync` int(1) NOT NULL,
        `Timestamp` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
            ON UPDATE CURRENT_TIMESTAMP,
```

```
    PRIMARY KEY (`CodNotificacion`),
    KEY `FK_NOTIFICACION_USUARIO` (`CodUsuario`),
    KEY `FK_NOTIFICACION_OFICINA` (`CodOficina`),
    CONSTRAINT `FK_NOTIFICACION_OFICINA` FOREIGN KEY (`CodOficina`)
        REFERENCES `oficina` (`CodOficina`) ON DELETE CASCADE
    ON UPDATE CASCADE,
    CONSTRAINT `FK_NOTIFICACION_USUARIO` FOREIGN KEY (`CodUsuario`)
        REFERENCES `usuario` (`CodUsuario`) ON DELETE CASCADE
    ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=23 DEFAULT CHARSET=utf8
    COLLATE=utf8_spanish_ci;
```

Código 4.6: [Modelo de datos] Sentencia SQL de creación de la tabla *Notificación.*

- **Tabla Localización:** En esta tabla se almacenan todas las localizaciones, expresadas en coordenadas de latitud y longitud, que se envían al servidor desde los dispositivos móbiles de los conductores. Estas coordenadas son las utilizadas por el sistema para la representación de las rutas en mapas.

```
CREATE TABLE `localizacion` (
    `CodLocalizacion` int(11) NOT NULL AUTO_INCREMENT,
    `CodUsuario` varchar(5) COLLATE utf8_spanish_ci NOT NULL,
    `Latitud` varchar(40) COLLATE utf8_spanish_ci DEFAULT NULL,
    `Longitud` varchar(40) COLLATE utf8_spanish_ci DEFAULT NULL,
    `Fecha` varchar(8) COLLATE utf8_spanish_ci DEFAULT NULL,
    `Hora` varchar(6) COLLATE utf8_spanish_ci DEFAULT NULL,
    PRIMARY KEY (`CodLocalizacion`),
    KEY `FK_LOCALIZACION_USUARIO` (`CodUsuario`),
    CONSTRAINT `FK_LOCALIZACION_USUARIO` FOREIGN KEY (`CodUsuario`)
        REFERENCES `usuario` (`CodUsuario`) ON DELETE CASCADE ON UPDATE
        CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=6593 DEFAULT CHARSET=utf8
    COLLATE=utf8_spanish_ci;
```

Código 4.7: [Modelo de datos] Sentencia SQL de creación de la tabla *Localización.*

Se ha incluido un campo `TIMESTAMP` en las tablas como medida para una mejor gestión de la información que contienen las mismas. Un ejemplo claro sería la investigación de posibles incidencias, tarea en la que sería de gran utilidad no solo conocer el contenido de los campos de los registros, sino también en que momento u orden cronológico se almacenaron.

4.1.5. Decisiones de diseño

Esta sección tiene como fin explicar las principales decisiones de diseño tomadas para el desarrollo de la herramienta.

Sincronizaciones

La herramienta desarrollada en este proyecto será usada por personas que cambian constantemente de localización: los transportistas se mueven continuamente de un destino a otro de su hoja de ruta. Los dispositivos en los que estará instalada la aplicación harán uso de su conexiones por radio para la comunicación con el servidor. Dichos dispositivos serán siempre los que abran las conexiones, debido a que no son terminales que tengan una dirección IP a la que el servidor pueda conectarse en el momento que lo necesite. Una vez iniciada la comunicación por parte del dispositivo móvil, existiría la posibilidad de que dicha conexión se mantuviera abierta de forma permanente para que el servidor pudiera enviar actualizaciones de los datos al terminal. Esta es la forma en la que funcionan por ejemplo las llamadas aplicaciones *push*.

La tecnología *push* tiene la ventaja de que el cliente recibe actualizaciones por parte del servidor de forma inmediata. Además, al mantener una conexión ya abierta, la comunicación se efectúa de forma más rápida. Sin embargo, desarrollar este tipo de sistemas lleva gran esfuerzo en cuanto a tiempo y recursos, por lo que hay que valorar si es realmente necesaria su implementación.

La otra posibilidad es el uso de tecnología *pull*, la cual se basa en que sea el terminal el que pregunte por las actualizaciones al servidor cuando sea necesario sin mantener ninguna conexión abierta. Ésta será la tecnología usada en el desarrollo de esta herramienta.

La aplicación *TMM* tiene un mayor porcentaje de envío de información al servidor que de recepción de datos: debe enviar su posición geográfica cada cierto tiempo, además de los reportes en las expediciones tratadas. Otro motivo más para emplear esta tecnología es que la dinámica de funcionamiento en estos procesos indica que, una vez que el transportista inicia su actividad, serán muy pocas las actualizaciones en los datos que recibirá por parte del servidor. Una ventaja que tendrá el empleo de esta tecnología es que se podrá programar la aplicación para que realice comunicaciones periódicas con el servidor con períodos configurables por los usuarios. De esta forma, las sincronizaciones podrán tener una mayor o menor frecuencia dependiendo de las necesidades de la empresa o incluso del usuario en concreto.

Lo siguiente que hay que tener en cuenta una vez que se ha decidido emplear la tecnología *pull* es que las aplicaciones en los dispositivos móviles pueden tener dos modos de funcionamiento dependiendo del uso de la conexión:

- **Modo *online*:** Se considera que una aplicación trabaja en este modo cuando ésta necesita poder disponer en cualquier momento de conexiones hacia el servidor, de forma que el interfaz de usuario se ve afectado en gran medida por estas conexiones. Un ejemplo sería una aplicación que muestra datos en pantalla que descarga del servidor en el momento de la visualización. Las ventajas de este modo son que la aplicación cliente es más ligera. La gran desventaja es que la aplicación puede quedar inutilizable si se pierde la connectividad. En cuanto a la navegación, puede transcurrir de manera más o menos fluida dependiendo de la cantidad de datos a descargar del servidor.

- **Modo *offline*:** Una aplicación trabaja en modo *offline* si no necesita tener conectividad con el servidor en todo momento, de forma que no queda bloqueada en caso de que dicha conectividad no exista en determinados momentos. La desventaja de que las aplicaciones funcionen en este modo es que se necesita almacenar y manejar más información en el dispositivo móvil, lo cual a veces puede traer complicaciones dada la naturaleza limitada en cuanto a recursos, proceso, etc. que tienen estos dispositivos. El cliente por tanto resulta un poco más pesado, aunque por otra parte, se consigue una mayor independencia de la aplicación con respecto a la conexión con el sistema.

Se ha decidido que el módulo cliente para Android funcione en modo *offline* debido a la necesidad por parte de los usuarios de no ver interrumpidas sus actividades por carencias de la herramienta. Es decir, si la aplicación funcionara en modo *online* y se perdiera conectividad, el transportista debería parar de trabajar y esperar a que la herramienta volviera a funcionar con todas sus capacidades. Esta situación provocaría un menor rendimiento y baja productividad de los empleados, efecto contrario al objetivo de la herramienta.

En cuanto a las desventajas que implicaría usar este modo en este caso en concreto, la cantidad de datos a almacenar en el dispositivo no es elevada, por lo que no se verá afectado el rendimiento de la aplicación.

Para conseguir que el cliente funcione en modo *offline* se ha diseñado un mecanismo de sincronizaciones entre el dispositivo y el servidor que permita mantener los datos

actualizados en ambos extremos en la medida de lo posible. La aplicación hace dos tipos de sincronizaciones:

1. Sincronización en primer plano: La aplicación hace conexiones con el servidor mientras se avisa al usuario de la operación. La forma de informar al usuario sobre esta sincronización es a través de un diálogo de progreso, el cual desaparece de la pantalla automáticamente al término de las operaciones. Este tipo de sincronizaciones ocurren de tres formas en la aplicación:
   - Al inicio de sesión: Mientras la aplicación valida al usuario y descarga los datos relativos a sus expediciones y notificaciones.
   - Al cerrar la aplicación por parte del usuario: Antes de cerrar la aplicación y borrar todos los datos del dispositivo se lleva a cabo una sincronización con el fin de enviar al servidor las últimas modificaciones realizadas sobre los datos para evitar que se pierdan.
   - El usuario puede forzar en cualquier momento una sincronización en primer plano a través de uno de los items de uno de los menús.
2. Sincronización en *background*: Cuando el usuario inicia sesión en la aplicación se programan sincronizaciones periódicas en segundo plano. Las modificaciones sobre los datos de expediciones y notificaciones que realiza el usuario se almacenan en el dispositivo. De la misma forma, las coordenadas geográficas que se van obteniendo cada cierto tiempo también son almacenadas en la base de datos de la aplicación. La función de estas sincronizaciones es enviar al servidor esos datos almacenados posteriormente a la sincronización anterior, así como recibir posibles actualizaciones de la parte servidora. El período entre sincronizaciones será configurable, de forma que puede ser adaptado a las necesidades concretas del escenario. Estas sincronizaciones se realizan de forma transparente para el usuario.

Mapas

Para la creación de mapas en la aplicación se emplean los mapas proporcionados por Google tanto en el cliente Android como en el perfil web. Google tiene disponibles varias opciones para la generación de mapas: APIs para Javascript, Flash, *Earth* y mapas estáticos, además de servicios web.

Para la integración de mapas en el navegador del perfil de gestión se ha decidido usar el API de Google Maps para Javascript. Esta decisión se ha tomado teniendo en cuenta el hecho de que este módulo está basado en servlets y páginas JSP. Aunque su implementación es más complicada que la opción de insertar mapas estáticos en la

página, proporciona diversas utilidades para manipular los mapas y añadirles contenido mediante diversos servicios, lo que permite conseguir una aplicación de mapas más sólida y completa.

En cuanto al cliente móvil, Google proporciona una librería externa específica para la representación de mapas en Android de forma nativa. Esta librería incluye el paquete `com.google.android.maps` , cuyas clases ofrecen entre otras opciones una variedad de controles para manejo del mapa.

Es importante destacar que, al ser este un proyecto final de carrera, en este caso se cumplen los términos de las condiciones del servicio del API de Google Maps. Una de esas condiciones indica que los mapas generados deberán ser públicos y gratuitos. Si este proyecto se llevara a la práctica, los mapas generados serían de uso interno para la empresa y por tanto debería usarse el API de Google Maps edición *Premier* [36].

Sesión continua en el cliente Android

Por seguridad y por que los datos que se mostrarán en la aplicación móvil son específicos y distintos para cada usuario, se quiere implementar un sistema de inicio de sesión. Por una parte sólo personas autorizadas podrán hacer uso de la herramienta y por otra sirve al sistema para conocer qué datos debe enviar al dispositivo.

Es necesario que la aplicación esté corriendo durante toda la actividad del usuario, puesto que debe enviar continuamente coordenadas geográficas al sistema, entre otras funciones. Por otra parte, sería muy pesado para el transportista tener que estar iniciando sesión cada vez que inicie la aplicación. Es por esto que una vez el usuario inicie sesión, la aplicación continuará ejectutándose hasta que sea el propio usuario el que decida salir de la aplicación. La herramienta seguirá comunicándose con el servidor aunque el conductor no esté usando el terminal y la pantalla esté bloqueada. El salir de la aplicación implica que, además de detener su actividad, se eliminarán todos los datos de la misma almacenados.

Servicios web

A la hora de implementar servicios web, existen dos filosofías principalmente: REST y SOAP. Desde que REST salió a la luz, siempre ha habido un debate en torno a su comparación con SOAP.

La arquitectura REST es sencilla, precisamente ese es su atractivo principal, de forma que fue rápidamente catalogado como alternativa a SOAP. Aún así, actualmente SOAP todavía posee el monopolio en el mundo de los servicios web. Ambos difieren en muchos aspectos comenzando por que REST fue concebido en el ámbito académico y SOAP es un estándar de la industria creado por un consorcio [37].

Las principales diferencias en el funcionamiento de ambos son:

- SOAP es un estilo de arquitectura orientado a RPC (Remote Procedure Call), es decir, un estilo basado en llamadas a procedimientos remotos, mientras que para REST solamente existen los métodos de HTTP y está orientado a recursos.
- REST no permite el uso estricto de "sesión"puesto que por definición es sin estado, mientras que para SOAP, al ser orientado a RPC, los servicios web crean sesiones para invocar a los métodos remotos.
- SOAP utiliza HTTP como un túnel por el que pasa sus mensajes, se vale de XML para encapsular datos y funciones en los mensajes. Si dibujásemos una pila de protocolos, SOAP iría justo encima de HTTP, mientras que REST propone HTTP como nivel de aplicación.

Ambas opciones presentan ventajas y desventajas, y es por ello que la comunidad de programadores está dividida entre ambas filosofías. La mayoría de las veces la decisión sobre qué tipo de servicios web implementar se basa en el coste que conlleva desarrollarlos en cada una de las opciones.

Los servicios web desarrollados en este proyecto serán de tipo SOAP. La implementación de los mismos resultará más sencilla haciendo uso de la librerá *kSOAP2*, la cual será explicada más en detalle en la siguiente sección.

4.2. Desarrollo e implementación

Una vez expuestos los objetivos principales de la aplicación, así como su funcionalidades y arquitectura, en esta sección se expondrán los aspectos relativos al desarrollo.

El proyecto se ha implementado sobre un sistema operativo Windows XP. En primera instancia, se mencionarán todas las herramientas que han sido utilizadas para el desarrollo del proyecto. Son las siguientes:

- JDK 1.6 de Java. Son las utilidades básicas y necesarias para la programación en el lenguaje Java.
- SDK de Android. Incluye las herramientas necesarias para el desarrollo de aplicaciones en la plataforma Android.
- Eclipse v3.4 Ganymede. Este es el IDE seleccionado para el desarrollo de todo el código de la aplicación, tanto del perfil móvil como de los perfiles web. Eclipse es una herramienta de código abierto multiplataforma que facilita mucho la tarea de desarrollo al programador con características como editor de texto con resaltado de sintaxis, distintas perspectivas, asistentes para creación de proyectos, clases, etc. Además, mediante plugins puede integrarse con otras herramientas.
- Plugin ADT de Android. Este plugin proporcionado por Google se integra con Eclipse, de forma que se pueden utilizar las herramientas de desarrollo para Android desde Eclipse.
- Apache Tomcat v6.0.18. Este es el servidor web utilizado para el funcionamiento de los perfiles web, así como de los servicios web expuestos para uso del cliente móvil.
- MySQL 5.1 Server. Es un sistema de gestión de base de datos relacional, multihilo y multiusuario.
- Toad for MySQL 4.1 Es una aplicación informática empleada para el desarrollo y administración de bases de datos relacionales usando SQL.
- Conector MySQL Java 5.1.7. Conector para la base de datos MySQL.

A continuación se comentará el desarrollo realizado para cada módulo de la herramienta. Las secciones de códigos adjuntadas con el texto han sido modificadas en algunos casos. Se han eliminando líneas del código original de los ficheros por ser prescindibles sobre lo que se pretende explicar y se han reestructurado de forma que aporten una mayor claridad en la explicación.

4.2.1. Servicios web

A la hora de construir servicios web, se pueden seguir dos tipos de estrategias:

- Estrategia *top-down*: Al diseñar un sistema se comienza pensando en una visión global de como va a funcionar todo. Luego se definen cuales van a ser los grandes componentes del sistema y poco a poco se va refinando y definiendo las funciones de partes más y más pequeñas hasta que se termina construyendo el sistema entero. La

construcción de servicios web con esta estrategia implica tener creado un documento WSDL de definición del servicio a partir del cual se crearía toda la implementación.

- Estrategia *bottom-up*: En un sistema diseñado con esta estrategia, se empieza por las partes más pequeñas, comenzando por los detalles sin conocer la visión global. Según se van definiendo soluciones para diversos problemas pequeños, estas soluciones se van conectando y va surgiendo una solución para varios de los subproblemas. Conforme se van subiendo niveles el sistema general va emergiendo de forma natural.

Para la creación de los servicios web de esta herramienta se ha seguido una estrategia *bottom-up*, es decir, primero se han implementado las clases .java necesarias, y a partir del código desarrollado se ha creado el servicio web y el fichero .wsdl. Además, el *runtime* usado es Apache Axis sobre un servidor Apache Tomcat v6, como ya se ha explicado anteriormente.

A continuación se explican los aspectos más relevantes de las clases implementadas:

- Clase *Conexion.java*: Su única función es la de abrir y cerrar conexiones con la base de datos. Lo más destacable son las siguientes líneas:

```
Class.forName("com.mysql.jdbc.Driver");
Connection con =
   DriverManager.getConnection("jdbc:mysql://localhost:3306/tmm", user ,
        password );
```

Código 4.8: [Servicio web] Creación de la conexión con la base de datos.

La primera instrucción carga el driver de conexión a la base de datos en MySQL. Dentro del conector se encuentra el archivo *Driver.class* en el interior de las carpetas com/mysql/jdbc.

Con la segunda línea se crea la conexión a la base de datos. La URL para la conexión se compone de tres partes separadas por ":", *jdbc:nombre_dbms:datos_de_conexión*. El valor *tmm* de la URL se corresponde con el nombre de la base de datos creada previamente.

- Clase *Consulta.java*: Es la clase que realiza todas las consultas a la base de datos. Contiene métodos para obtener y almacenar datos referentes a las expediciones y notificaciones. También almacena las coordenadas de posición de los usuarios, además de obtener datos referentes a éstos.

Las operaciones realizadas en esta clase quedan reflejadas en *logs*, los cuales no solo son de gran utilidad, sino prácticamente necesarios en caso de investigación de incidencias o anomalías. El código que implementa esta funcionalidad es el siguiente:

```
private FileHandler hand;
private Logger log;
    ...
public Consulta()
{
    ...
    try
    {
        // Creación del manejador para los ficheros.
        hand = new FileHandler("tmmLog.txt", 1000000, 1, true);
        hand.setFormatter(new SimpleFormatter());
    }
    catch (IOException ioe){}

  // Encuentra el logger uc3m.tmm o lo crea si no existe
    log = Logger.getLogger("uc3m.tmm");
    log.addHandler(hand);
}
    ...
```

Código 4.9: [Servicio web] Implementación del mecanismo de *logs*.

Los ficheros de *log* son creados y escritos de forma cíclica. El constructor recibe por parámetro el número de ficheros se desean usar para almacenar la información y cada vez que se alcanza el tamaño máximo, definido también por parámetro, se crea un fichero nuevo. Una vez se hayan creado y escrito el número de ficheros especificado, se sobreescribirá el primero de ellos, después el segundo y etc. El tamaño de los ficheros se indica en *bytes*.

La escritura en los ficheros se realiza a través del método *logp*, al que se le indica un identificador de nivel, la clase y método desde el que se escribe y el texto que se quiera dejar reflejado. En este proyecto se han usado los niveles *INFO* para reflejar datos meramente informativos y *SEVERE* para registrar errores en la actualización de datos en la base de datos, así como excepciones de tipo SQL producidas en las operaciones con la misma.

A continuación se muestra un ejemplo de la información escrita en el *log*:

```
21-ago-2010 10:52:44 Consulta.java getEntregas
INFO: Nº de entregas a enviar: 9
21-ago-2010 10:52:45 Consulta.java getRecogidas
INFO: Nº de recogidas a enviar: 9
21-ago-2010 10:52:47 Consulta.java getNotificaciones
INFO: Nº de notificaciones a enviar: 7
...
21-ago-2010 10:54:49 Consulta.java setLocalizaciones
INFO: Nº de localizaciones recibidas: 2
21-ago-2010 10:54:49 webServices.pfc.Consulta
setLocalizaciones
INFO: -- OK -- u0013 -- 6
21-ago-2010 10:54:49 webServices.pfc.Consulta
setLocalizaciones
INFO: -- OK -- u0013 -- 7
21-ago-2010 10:55:48 Consulta.java setEntregas
INFO: Nº de entregas recibidas: 1
21-ago-2010 10:55:48 webServices.pfc.Consulta setEntregas
INFO: -- OK -- u0013 -- 7 -- 2 -- Entrega dañada -- Daños
21-ago-2010 10:55:48 Consulta.java getEntregas
INFO: Nº de entregas a enviar: 0
21-ago-2010 10:55:48 Consulta.java getRecogidas
INFO: Nº de recogidas a enviar: 0
21-ago-2010 10:55:49 Consulta.java getNotificaciones
INFO: Nº de notificaciones a enviar: 0
21-ago-2010 10:55:49 Consulta.java setLocalizaciones
INFO: Nº de localizaciones recibidas: 2
21-ago-2010 10:55:49 webServices.pfc.Consulta
setLocalizaciones
INFO: -- OK -- u0013 -- 8
21-ago-2010 10:55:49 webServices.pfc.Consulta
setLocalizaciones
INFO: -- OK -- u0013 -- 9
...
21-ago-2010 11:05:48 Consulta.java setEntregas
INFO: Nº de entregas recibidas: 1
21-ago-2010 11:05:48 webServices.pfc.Consulta setEntregas
GRAVE: -- ERR -- u0013 -- 5 -- 2 -- Daños
```

La información que se añade en cada caso según el tipo de elemento es:

- Entregas y recogidas: Se especifica el código de usuario, el código del elemento en cuestión, el estado con el que se reporta y, en su caso, observaciones del

transportista e incidencias.

- Notificaciones: En este caso se especifica el código de usuario, el código de la notificación y el estado.
- Localizaciones: Para las localizaciones solo se incluye el código del usuario y el de la localización.

En las últimas líneas del *log* anterior se ha incluido un ejemplo de error al realizar un reporte de una entrega. En este ejemplo se ilustra claramente la utilidad de estos ficheros: aunque el reporte realizado por el transportista no queda reflejado en la base de datos, se podrá averiguar si llegó a realizarse, si se entregó o no la entrega y si hubo alguna incidencia relacionada con la misma.

- Las clases del modelo de datos son *Delivery.java*, *Collection.java*, *Message.java* y *Location.java*. Estas clases son *beans* que encapsulan los campos de las diferentes tablas de la base de datos.
- Por último, la clase *Service.java* contiene los métodos expuestos por el servicio web. Dichos métodos son los siguientes:

```
// Valida un usuario en el sistema
public String validarUsuario(String codUsuario, String password){}

// Devuelve los datos del usuario de la primera sincronización
public String resetUsuario(String codUsuario){}

// Devuelve las entregas asociadas a la hoja de ruta de un usuario
public Delivery[] getEntregas(String codUsuario){}

// Almacena los reportes de entregas hechos por el usuario
public String setEntregas (String codUsuario, Delivery entregas[]){}

// Devuelve las recogidas asociadas a la hoja de ruta de un usuario
public Collection[] getRecogidas(String codUsuario){}

// Almacena los reportes de recogidas hechos por el usuario
public String setRecogidas (String codUsuario, Collection recogidas[]){}

// Devuelve las notificaciones asociadas a un usuario
public Message[] getNotificaciones (String codUsuario){}

// Almacena los reportes de notificaciones hechos por el usuario
public String setNotificaciones
    (String codUsuario, Message notificaciones[]){}
```

```
// Almacena las coordenadas enviadas por el usuario sobre su posición
public String setLocalizaciones
    (String codUsuario, Location localizaciones[]){}
```

Código 4.10: [Servicio web] Métodos expuestos por el servicio.

4.2.2. Cliente Android

En esta sección se comentarán los aspectos más importantes o destacables del desarrollo de la herramienta sobre la plataforma Android. Para que el lector pueda generar en su mente una visión global de la aplicación que facilite el seguimiento de la explicación sobre su desarrollo, se muestra en la figura 4.3 el diagrama de navegación de la herramienta.

Con el mismo objetivo que el perseguido con el diagrama de navegación de pantallas, se presenta a continuación una descripción de las clases .java creadas en este proyecto y su propósito. Dichas clases se agrupan en los paquetes:

- Paquete *uc3m.TMMCliente*: En este paquete se encuentran las clases:
  - *BD.java*: Clase que gestiona todas las operaciones relacionadas con la base de datos SQLite.
  - *Pfc.java*: Clase que contiene como atributos la base de datos, un *parser* y un *flag* indicando si el usuario ha iniciado sesión en el sistema.
  - *Parser.java*: Esta clase contiene los métodos necesarios para empaquetar la información que será enviada al servidor en objetos *SoapObject*, así como para procesar la información recibida por el mismo.
  - *Utilities.java*: Esta clase contiene métodos de configuración del idioma en el dispositivo, de manejo de sincronizaciones y de creación de notificaciones.
- Paquete *uc3m.TMMCliente.components*: En este paquete se agrupan clases que ha sido necesario personalizar para el cumplimiento de alguna funcionalidad de la aplicación.
  - *CustomAdapterCollections.java, CustomAdapterDeliveries.java, CustomAdapterMessages:* Estas clases heradan de *BaseAdapter.java* y sirven para personalizar la apariencia de los listados de recogidas, entregas y notificaciones, respectivamente.

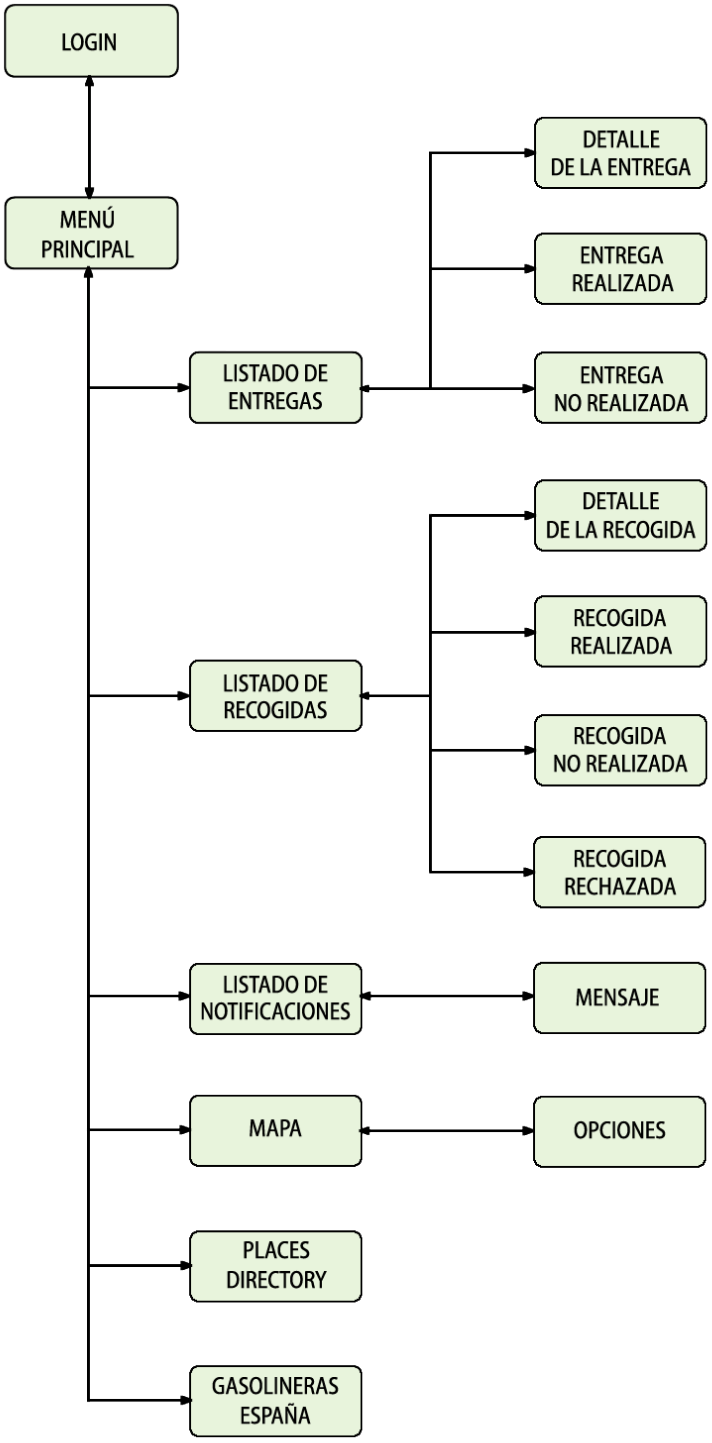

Figura 4.3: [Aplicación Android] Diagrama de navegación.

  - *TMMOverlayItem*: Clase creada para agrupar la información referente a un elemento que se muestra en el mapa.

- Paquete *uc3m.TMMCliente.model*: Las clases que conforman el modelo de datos se agrupan en este paquete.

  - *Delivery.java, Collection.java, Message.java*: Son los *beans* de los elementos entrega, recogida y notificación, respectivamente.

- Paquete *uc3m.TMMCliente.screens*: Contiene todas las clases que representan pantallas de la aplicación. Éstas son en total quince clases que heredan de *Activity*, más una clase que hereda de *Overlay.java* y que representa una capa del mapa.

- Paquete *uc3m.TMMCliente.sync*: Las clases principales implicadas en las sincronizaciones se empaquetan en este directorio.

  - *SyncService.java:* Esta clase es un servicio que se ejecuta en segundo plano y que hace posible las sincronizaciones, así como la recogida de las coordenadas del usuario.
  - *SyncTask.java:* Tarea programada que inicia las sincronizaciones.
  - *SyncHelper.java:* Hilo que realiza las comunicaciones de red.

- Paquete *uc3m.TMMCliente.track*: En este paquete se incluye una única clase, llamada *Track.java*, la cual obtiene y almacena las coordenadas geográficas del usuario.

A continuación se exponen la implementación de funcionalidades determinadas de la aplicación. Las capturas de pantalla añadidas para ilustrar las explicaciones de este módulo han sido realizadas sobre un dispositivo real, un *smartphone* Motorola Milestone con sistema operativo Android 2.1.

Sincronizaciones

La aplicación realizará conexiones con el servidor para recibir y enviar los datos asociados al usuario. El tipo de sincronizaciones que se dan en el cliente móvil se explicaron en la página 68, pero en esta sección se expondrá como desarrollar mediante código la funcionalidad descrita.

Hay principalmente tres clases implicadas en el mecanismo de sincronización:

- Clase *SyncService.java*: Dado que las sincronizaciones deben producirse incluso cuando el dispositivo tiene la pantalla bloqueada, es necesario hacer uso de un

componente *Service*, el cual continuará activo en segundo plano en todo momento. Este servicio se activa en el momento en el que el usuario inicia sesión y se termina su ejecución cuando el usuario cierre sesión en la aplicación. La función de este servicio consiste en crear una tarea programada mediante un temporizador, la cual será la encargada de activar la sincronización cuando corresponda. Además, el servicio posee una instancia de la clase encargada de almacenar las coordenadas del dispositivo para ser enviadas al servidor posteriormente. La razón por la que esta instancia pertenece al servicio es porque las coordenadas también deben ser almacenadas aunque el dispositivo se encuentre con la pantalla bloqueada.

```
public class SyncService extends Service
{
    // Temporizador
    Timer syncTimer;
    // Clase que almacena las coordenadas del usuario
    Track track;
    ...

    @Override
    public void onCreate()
    {
        super.onCreate();
        ...

        // Creación del temporizador
        syncTimer = new Timer();
        // Programación de la tarea en intervalos de minutos
        syncTimer.schedule(new SyncTask(this), minsSync*60000,
                                               minsSync*60000);
        // Creación de una instancia de la clase Track
        track = new Track(this.getApplicationContext());
    }
    ...
}
```

Código 4.11: [Aplicación Android] Clase SyncService.java.

Como componente de Android que es un servicio, debe declararse en el fichero AndroidManifest.xml. El servicio es activado como componente desde la *activity* del menú principal mediante un *intent*. Éste lleva asociada una acción propia de la herramienta definida como *uc3m.TMMCliente.SYNC_SERVICE*. Es por ello que

en el fichero AndroidManifest.xml la declaración del servicio debe definirse de la siguiente forma:

```
<service android:name=".sync.SyncService">
    <intent-filter>
        <action android:name="uc3m.TMMCliente.SYNC_SERVICE" />
    </intent-filter>
</service>
```

Código 4.12: [Aplicación Android] AndroidManifest.xml: Declaración del servicio.

- Clase *SyncTask.java*: El código de esta clase es muy breve debido a que la sincronización está muy modularizada. Existe un método *sincronizar()* en la clase *Utilities.java* que es el encargado de iniciar el hilo que realiza las operaciones de red. A su vez este método es invocado por métodos que gestionan diálogos en pantalla, notificaciones a mostrar, etc. dependiendo de si la sincronización es la inicial, en segundo plano o está forzada por el usuario. En el caso concreto de esta clase *SyncTask*, el método en concreto se llama *sincronizarEnBG()*, y además de invocar la sincronización, también lanza notificaciones en el caso en que se hayan recibido nuevas recogidas, así como se ocupa de refrescar las pantallas con la nueva información recibida.

  Por tanto, esta clase hereda de *TimerTask.java*, posee un atributo que es una instancia de la clase *Utilities.java* y su método *run()* consta de una única línea, la cual es una llamada al método *sincronizarEnBG()*.

- Clase *SyncHelper.java*: En esta clase se crean todas las peticiones a enviar al servidor, se realizan las conexiones y se procesan las respuestas recibidas. Como no podía ser de otra forma, la clase es un hilo ya que las operaciones de red tardan un tiempo imposible de determinar y se deben evitar posibles bloqueos en la aplicación durante este período.

  Para la comunicación con el servidor se ha hecho uso de *kSoap2*, que es una librería para clientes de web services enfocados a entornos Java con limitaciones, como Applets o aplicaciones J2ME [38]. A continuación se incluye parte del código de validación del usuario como ejemplo de uso de esta librería. En este método se definen parámetros necesarios para la comunicación, como la dirección del servicio o la operación a realizar, se contruye la petición SOAP, se realiza el envío y se obtiene la respuesta. El método que hace efectivo el enío de las peticiones se adjunta también a continuación del siguiente:

```
private void validateUser ()
{
    String serviceAddress =
              "http://X.X.X.X:8080/TMMWebServices/services/Service";
    String ptoOperName = "validarUsuario";
    String soapAction = serviceAddress+ptoOperName;
    // Creación de la petición SOAP
    SoapObject request = new SoapObject(targetNamespace, ptoOperName);
    // Inclusión del código de usuario y contraseña en la petición
    request.addProperty("codUsuario", tmm.getBD().fetchUser().get(0));
    request.addProperty("password", tmm.getBD().fetchUser().get(1));

  // Envío de la petición y recepción de la respuesta
    SoapObject result = makeSoapConnection(request);


    if (!connectionError)
    {
        // Procesado de la respuesta para obtener la información
        String validateUserResult =
         result.getProperty("validarUsuarioReturn").toString();
     ...
    }
}
```

Código 4.13: [Aplicación Android] Uso de kSoap2: Método *validateUser()*.

A la hora de desarrollar una aplicación sobre esta plataforma, es habitual que cliente y servidor se encuentren en la misma máquina. Un error muy usual es utilizar la dirección de *loopback* 127.0.0.1 de la máquina en la variable *serviceAddress* mostrada en el código anterior. En Android, cada instancia del emulador corre detras de un servicio de *router/firewall* virtual que lo aisla de los interfaces de red de la máquina, así como de internet. Un dispositivo virtual no puede ver la máquina de desarrollo u otros dispositivos virtuales de la red. La dirección equivalente a 127.0.0.1 es la 10.0.0.2 y, por tanto, esta será la dirección a usar en su lugar.

A continuación se incluye el método que envía las peticiones de la herramienta al servidor. En el código del proyecto, algunas variables como por ejemplo la dirección del servicio son globales para esta clase *SyncHelper*. Distintos métodos en la clase construyen los objetos *SoapObject* dependiendo de la operación a realizar y todos ellos usan este método *makeSoapConnection* para el envío de dichas peticiones.

```
public SoapObject makeSoapConnection (SoapObject request)
{
    SoapObject result = null;
    String targetNamespace = "http://pfc.webServices";

    // Clase de kSoap2 de conexión HTTP basada en J2SE
    HttpTransportSE httpt = new HttpTransportSE(serviceAddress);
    // Envelope de SOAP versión 1.1
    SoapSerializationEnvelope envelope =
                    new SoapSerializationEnvelope( SoapEnvelope.VER11 );
    // Comportamiento estándar del envelope
    envelope.dotNet = false;
    // Inclusión de la petición en el envelope
    envelope.bodyOut = request;
    try
    {
        // Envío de la petición
        httpt.call(soapAction, envelope);
        // Recepción de la respuesta
        result = (SoapObject)envelope.bodyIn;
    }
    catch(Exception e)
    {
        connectionError = true;
        e.printStackTrace();
    }
    return result;
}
```

Código 4.14: [Aplicación Android] Uso de kSoap2: Método *makeSoapConnection()*.

Las variables referentes a la conexión con el servidor, como el *targetNamespace, seviceAddress*, etc. vienen especificadas en el fichero .wsdl del servicio web. A la hora de manejar los *SoapObject* tanto de la petición como de la respuesta es necesario tener presente el fichero .wsdl para conocer la estructura interna de los mismos.

Se adjunta a continuaciós los ficheros .xml generados en el intercambio de datos de la operación de validación del usuario. El primero de ellos es la petición realizada por el cliente Android. En ella se envían el código de usuario y la contraseña introducidos por el transportista. El segundo fichero es la respuesta recibida del servidor. Como la validación ha sido correcta, el valor que es devuelto en el elemento *validarUsuarioReturn* es un número que indica los minutos que forman el período entre sincronizaciones. En el

caso de que el usuario no fuera válido el servidor devolvería un código de error.

PETICIÓN

```
<v:Envelope xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:d="http://www.w3.org/2001/XMLSchema"
            xmlns:c="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:v="http://schemas.xmlsoap.org/soap/envelope/">
  <v:Header />
    <v:Body>
      <n0:validarUsuario id="o0" c:root="1"
                                 xmlns:n0="http://pfc.webServices">
        <codUsuario i:type="d:string">u0013</codUsuario>
        <password i:type="d:string">23456</password>
      </n0:validarUsuario>
  </v:Body>
</v:Envelope>
```

RESPUESTA

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <validarUsuarioResponse xmlns="http://pfc.webServices">
      <validarUsuarioReturn>5</validarUsuarioReturn>
    </validarUsuarioResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Por último, para que las sincronizaciones sean posibles, es necesario incluir el permiso *android.permission.INTERNET* en el fichero AndroidManifest.xml. En caso contrario, el dispositivo denegará a la aplicación el acceso a la red.

Notificaciones

En relación con las sincronizaciones se encuentran las notificaciones. Una de las funcionalidades de la aplicación es el aviso al usuario de la recepción en el dispositivo de nuevas recogidas. Para implementar dichos avisos se ha utilizado el mecanismo de notificaciones propio de Android, el cual incluye aviso sonoro, visual mediante el LED y vibración del dispositivo. Posteriormente, el usuario puede visualizar la notificación en la pantalla y, tras seleccionarla, se elimina dicha notificación y se muestra en pantalla el listado de recogidas.

El método *crearNotificacion()* de la clase *Utilities.java* contiene todo el código necesario para la implementación de esta funcionalidad, por lo que se adjunta el mismo a modo explicativo. Básicamente consiste en crear un objeto *Notification*, configurar sus parámetros y registrar dicha notificación en el *NotificationManager*:

```
private void crearNotificacion()
{
    // Icono que aparecerá en la barra de estado
    int icon = R.drawable.lorrygreen64x64;
    // Titulo de la notificación
    CharSequence tickerText = "TMM";
    // Momento de la notificación
    long when = System.currentTimeMillis();

    // Creación de la notificación
    Notification notification = new Notification(icon, tickerText, when);

    // Configuración del parpadeo del LED
    notification.ledARGB = 0xff00ff00;
    notification.ledOnMS = 3000;
    notification.ledOffMS = 1000;
    notification.flags |= Notification.FLAG_SHOW_LIGHTS;

    // Configuración de la vibración y el sonido
    notification.defaults |= Notification.DEFAULT_VIBRATE;
    notification.defaults |= Notification.DEFAULT_SOUND;

    // Con este flag la notificación desaparece tras su
    // revisión por el usuario.
    notification.flags |= Notification.FLAG_AUTO_CANCEL;

    // Textos que se muestran en el desplegable de notificaciones
    CharSequence contentTitle = "TMM";
```

```
    CharSequence contentText = context.getString(R.string.NewCollects);

    // Acción que se realiza al seleccionar la notificación en el desplegable.
    // En este caso se abre la pantalla del listado de recogidas.
    Intent notificationIntent =
     new Intent(context, CollectionsListScreen.class);
    PendingIntent contentIntent =
     PendingIntent.getActivity(context, 0, notificationIntent, 0);
    notification.setLatestEventInfo(context, contentTitle, contentText,
        contentIntent);

    // Creación del objeto NotificationManager
    NotificationManager mNotificationManager =
    (NotificationManager)context.getSystemService(Context.NOTIFICATION_SERVICE
        );

    // Registro de la notificación en el NotificationManager
    mNotificationManager.notify(TMM_ID, notification);
}
```

Código 4.15: [Aplicación Android] Creación de notificaciones.

A continuación se incluyen dos capturas de pantalla: en la primera de ellas se muestra el aviso de notificación en la barra de estado del dispositivo, y en la segunda, el desplegable de notificaciones con los datos de la notificación lanzada.

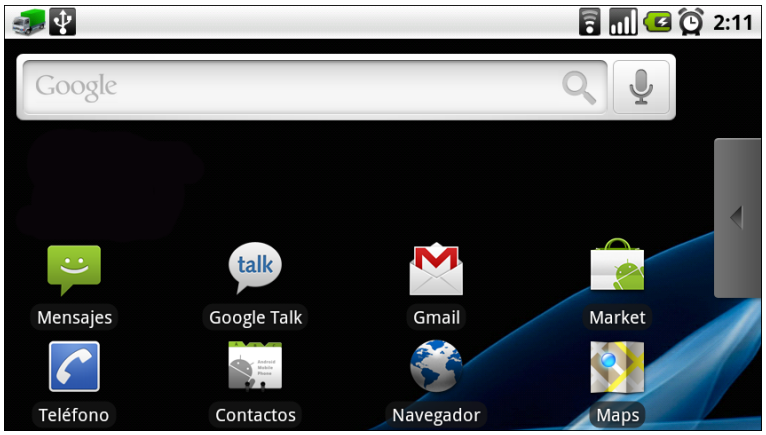

Figura 4.4: [Aplicación Android] Notificación en la barra de estado.

Figura 4.5: [Aplicación Android] Notificación en el desplegable de notificaciones.

Para que el dispositivo vibre en la recepción de una nueva recogida es necesario declarar el permiso *android.permission.VIBRATE* en el fichero AndroidManifest.xml.

Preferencias de idioma

El interfaz de usuario de la aplicación se puede visualizar en dos idiomas distintos, inglés y español. La selección del idioma puede realizarla el usuario desde la pantalla de inicio de sesión y desde la pantalla del menú principal. La forma de implementarlo es mediante la clase *Configuration*, la cual describe toda la información de configuración del dispositivo que puede tener un impacto sobre los recursos usados por la aplicación, como por ejemplo las preferencias de idioma del usuario.

El idioma seleccionado por el usuario no solo es configurado en la aplicación en el momento de su selección, sino que también es almacenado de forma persistente mediante un objeto de la clase *SharedPreferences*. Este objeto permite guardar pares clave-valor de tipos primitivos y se usa en general para almacenar todas las preferencias del usuario que pueden ser configurables en una aplicación. De esta forma, la aplicación recuerda y usa siempre estas preferencias sin necesidad de que el usuario tenga que seleccionarlas cada vez que use la herramienta.

En el siguiente extracto de código se muestra conjuntamente como configurar el idioma y almacenarlo en el objeto de preferencias:

```
public void setIdioma ( String idioma)
{
    // Obtención de las preferencias
    SharedPreferences settings = context.getSharedPreferences("MyPrefs", 0);
```

```
    // Idioma actual de la aplicación
    Locale locDefault = Locale.getDefault();

    // Idioma que el usuario ha seleccionado
    Locale loc = null;

    // Obtención del Locale correspondiente al idioma seleccionado
    // y almacenamiento en el objeto de preferencias
    SharedPreferences.Editor editor = settings.edit();
    if (idioma.equals("es")){
        loc = new Locale("es", "ES");
        editor.putString("idioma", "es" );
    }else{
        loc = new Locale("en", "UK");
        editor.putString("idioma", "en" );
    }
    editor.commit();

    // Si el idioma actual es distinto del seleccionado por el
    // usuario se cambia la configuración con el nuevo idioma
    if ( ! locDefault.getLanguage().equals(loc.getLanguage()))
    {
        Locale.setDefault(loc);
        Configuration config = new Configuration();
        config.locale = loc;
        context.getResources().updateConfiguration(config, null);

        // Tras cambiar el idioma, se reinicia la pantalla para mostrarla
        // los cambios realizados
            Intent i = new Intent(RESTART_SCREEN);
            context.sendOrderedBroadcast(i, null);
    }
}
```

Código 4.16: [Aplicación Android] Configuración del idioma(I).

El cambio del idioma se mantiene a través de la navegación entre pantallas siempre y cuando no cambie la orientación del terminal. Si la pantalla rota hacia otra orientación, se produce un cambio de configuración y el idioma que se muestra pasa a ser el idioma por defecto del dispositivo aunque el usuario hubiera seleccionado otro. Para evitar este comportamiento, en la creación de todas las pantallas de la aplicación se recupera el idioma del objeto de preferencias y se configura con el método anterior. Así, cuando la

orientación cambia y se reinicia la *Activity*, se vuelve a configurar el idioma que el usuario eligió.

En la anterior sección de código queda ilustrada la forma en la que se almacenan valores en el objeto de preferencias, y a continuación se mostrará como se recuperan valores del mismo. Como el idioma es almacenado como un *String*, el método para recuperarlo es *getString(arg1, arg2)*, donde el primer parámetro es la clave del valor que se desea recuperar, y el segundo parámetro es el valor por defecto que se obtendrá en caso de no existir dicho par clave-valor en el fichero de preferencias. El código implementado en la creación de todas las pantallas es el siguiente:

```
// Obtención del objeto de preferencias
SharedPreferences settings = getSharedPreferences("MyPrefs", 0);
// Obtener el idioma guardado en las preferencias.
String idioma = settings.getString("idioma", "es");
// Método mostrado en el código anterior y que fija el idioma
utilities.setIdioma(idioma);
```

Código 4.17: [Aplicación Android] Configuración del idioma(II).

Actualización de pantallas

Las aplicaciones Android siguen una filosofía muy modular formada por componentes que se activan y desactivan entre sí. Esta filosofía se hace muy evidente para el programador experimentado en el desarrollo sobre otras plataformas que comienza a desarrollar para Android. Uno de los problemas más usuales con los que un programador se enfrenta en este sistema es el paso de información entre componentes, como por ejemplo entre un *service* y una *activity*.

En esta aplicación se hace necesario la comunicación entre el servicio de sincronización y la pantalla que se está visualizando en ese momento. Si el usuario está visualizando por ejemplo el listado de recogidas y en ese momento el dispostivo recibe nuevas recogidas mediante la sincronización en segundo plano, la pantalla debe actualizarse con la nueva información. La forma en la que se ha solucionado este problema ha sido mediante el uso de los componentes *BroadcastReceiver*. Este componente, del que se habló en la sección 2.4.1, notificará a las pantallas sobre el evento de nueva información recibida.

La aplicación usa componentes *BroadcastReceiver* no sólo para lanzar eventos tras las operaciones de sincronización de datos en el dispositivo sino también después de un cambio de idioma por parte del usuario. Estas funcionalidades afectan al interfaz

de usuario en varias de las pantallas de la aplicación. En concreto, la sincronización afecta a las pantallas *LoginScreen*, *MainMenuScreen*, *DeliveriesScreen*, *CollectionsScreen*, *MessagesScreen* y *MapScreen*. En cuanto al cambio de idioma, afecta a las pantallas desde las que se puede realizar esta operación, es decir, la pantalla *LoginScreen* y la pantalla *MainMenuScreen*, que deberán actualizar todos sus recursos para ser mostrados en el idioma seleccionado.

Estas dos funcionalidades de sincronización y cambio de idioma llevan asociadas distintas acciones a realizar por la *activity* que recoge el evento. Las *activities* registran filtros aplicables al *broadcast receiver* de forma que sólo sean notificadas de aquellas acciones definidas por el filtro. Las posibilidades son:

1. ACTION_VIEW: es una acción definida por el sistema Android, la cual será usada para definir la necesidad de una actualización en los datos mostrados por pantalla. Las pantallas que registran filtros para esta acción son aquellas que muestran información de los datos almacenados, es decir, las pantallas *MainMenuScreen*, *DeliveriesScreen*, *CollectionsScreen*, *MessagesScreen* y *MapScreen*. Los mensajes de *broadcast* enviados con esta acción se lanzan siempre tras las sincronizaciones y según la pantalla que los recibe, se modifican valores de elementos visuales o se reinicia la pantalla completa.

2. RESTART_SCREEN: es una acción definida en la aplicación que transmite la necesidad de reiniciar por completo la pantalla que se está visualizando, es decir, finalizar la *activity* y volverla a construir para mostrarla. Esto es necesario debido a que cierta información que se muestra al usuario sólo es configurable en la creación de una pantalla, de forma que no se puede modificar dinámicamente durante su visualización. La acción es realizada por las pantallas *LoginScreen* y *MainMenuScreen*, puesto que se lanza siempre tras un cambio de idioma.

3. FINISH: identifica la necesidad de cerrar una *activity* y por tanto la vista o pantalla asociada a ella. Las pantallas que filtran estos eventos son *LoginScreen* y *MainMenuScreen*. En el primer caso, porque se hace necesario cerrar la pantalla de inicio de sesión tras la primera sincronización, y en el segundo, para cerrar el menú principal tras la sincronización final previa a que el usuario cierre sesión en la aplicación.

A continuación se ilustra el uso del *BroadcastReceiver* sobre la pantalla del menú principal, que define un filtro con las tres posibles acciones descritas anteriormente.

Lo primero es declarar el componente en la clase y sobreescribir su método *onReceive*. Este método recibe como parámetro el contexto y un *Intent* donde viene la acción que describe la operación a realizar en cada caso:

```
public class MainMenuScreen extends Activity
{
    ...

    // Declaración del componente BroadcastReceiver
    private BroadcastReceiver receiver = new BroadcastReceiver()
    {
        @Override
        public void onReceive(Context context, Intent intent)
        {
            if (intent.getAction().equals(utilities.RESTART_SCREEN))
                reiniciarPantalla();
            else if (intent.getAction().equals(Intent.ACTION_VIEW))
                refreshScreen();
            else if (intent.getAction().equals(utilities.FINISH))
                finish();
        }
    };
    ...
}
```

Código 4.18: [Aplicación Android] Declaración de un *BroadcastReceiver*.

En segundo lugar, se debe registrar este componente en la *activity* en el inicio de la misma y desregistrarlo en su finalización, por eso se ha implementado este código en los métodos *onResume* y *onPause*:

```
public class MainMenuScreen extends Activity
{
    ...
 @Override
    protected void onResume()
    {
        super.onResume();

        // Definición de las acciones a filtrar por el BroadcastReceiver
        IntentFilter filter = new IntentFilter(Intent.ACTION_VIEW);
        filter.addAction(utilities.RESTART_SCREEN);
        filter.addAction(utilities.FINISH);

        // Registrar el BroadcastReceiver
        registerReceiver(receiver, filter);
```

```
        ...
    }

    @Override
    protected void onPause()
    {
        super.onPause();
        ...

        // Desregistrar el BroadcastReceiver
        this.unregisterReceiver(receiver);
    }
      ...
}
```

Código 4.19: [Aplicación Android] Registro de un *BroadcastReceiver*.

Gracias al código mostrado anteriormente, la pantalla del menú principal ya es capaz de recibir eventos lanzados por el *broadcast*. Sólo falta implementar el código que lanza dichos eventos, el cual consiste en una invocación al método *sendOrdererBroadcast*. Dicho método recibe un *Intent* como parámetro donde se define la acción y, opcionalmente, un permiso requerido por el recibidor para poder recibir el mensaje de *broadcast*.

```
// Definición del intent y su acción asociada
Intent i = new Intent(Intent.ACTION_VIEW);
// Envío del mensaje de broadcast, para el que no se requieren permisos
context.sendOrderedBroadcast(i, null);
```

Código 4.20: [Aplicación Android] Envío de un mensaje de *broadcast*.

Geolocalización

La herramienta usa geolocalización para dos funcionalidades diferentes. Por una parte, almacena periódicamente las coordenadas geográficas en las que se encuentra el usuario para enviarlas al servidor y, por otra parte, muestra en la pantalla del mapa la localización del usuario de forma actualizada.

Todas las clases necesarias para implementar la geolocalización se encuentran en el paquete *android.location*. Las clases e interfaces usadas en este proyecto son:

1. Clase *LocationManager*: esta clase da acceso a los servicios de localización del sistema.
2. Clase *Location*: representa una localización geográfica en un determinado momento. Define parámetros como la longitud, latitud, altitud, etc.
3. Clase *Criteria*: esta clase es usada para especificar los criterios de selección de los proveedores de localización.
4. Clase *LocationListener*: para obtener actualizaciones de la posición del terminal es necesario implementar este interfaz, el cual define métodos para detectar cambios en la ubicación o la disponibilidad del proveedor.

En primer lugar, se obtendrá una instancia de la clase *LocationManager* mediante el método *getSystemService*. Este método devuelve manejadores de diferentes servicios del sistema dependiendo del parámetro que reciba, en este caso concreto será el *string* asociado al servicio de localización, es decir, "LOCATION_SERVICE". Usando el objeto *LocationManager* se obtendrá un proveedor para la localización, además de registrar el período entre actualizaciones de la posición.

Existen varias formas de determinar el proveedor que será utilizado. Una forma es indicar directamente qué tipo de proveedor se desea usar, pudiendo especificarse "GPS_PROVIDER" si se desea obtener la localización mediante GPS, o bien especificando "NETWORK_PROVIDER" si se desea obtener usando la red. Estas constantes se pasarían como parámetro al método *requestLocationUpdates*, junto con el tiempo y la distancia mínimos entre actualizaciones. El último parámetro de este método es un objeto *LocationListener*.

También es posible definir unos parámetros mediante los cuales seleccionar un proveedor u otro según se ajusten más o menos a dichos parámetros. Para ello, se usa el método *getBestProvider* al que se le pasa como argumento un objeto de tipo *Criteria*. En este objeto se pueden definir los criterios en cuanto a precisión, rumbo, altitud, velocidad, requerimientos de energía y coste monetario. El método *getBestProvider* devuelve únicamente proveedores para los cuales la *activity* tiene permitido el acceso. Si se obtienen varios proveedores que cumplan con los criterios definidos, el método devuelve sólo aquel que mejor se ajuste a los criterios. Si, por el contrario, ningún proveedor se ajusta a los criterios definidos, éstos se ignoran en el siguiente orden:

1. Energía requerida
2. Precisión
3. Rumbo
4. Velocidad
5. Altitud

El criterio sobre si el proveedor está permitido incurrir en coste monetario no es ignorado en ningún caso.

En la aplicación *TMM*, se ha seguido este segundo método para la obtención del proveedor, puesto que representa una ventaja frente a la definición directa de un único proveedor en el caso de que dicho proveedor no se encuentre disponible. Si por ejemplo se definiera la obtención de la ubicación mediante GPS y éste no estuviera disponible, no podrían obtenerse las coordenadas del usuario. En cambio, mediante el segundo método se obtendría la ubicación del transportista usando redes móviles o incluso Wi-Fi.

Tras la obtención del proveedor, se implementa la interfaz *LocationListener* y los métodos que proporciona para obtener la localización.

En la aplicación se usa geolocalización tanto en la clase *MapScreen* para mostrar la ubicación del usuario en mapa, como en la clase *Track*, encargada de almacenar la posición geográfica de forma transparente para el usuario. A continuación se muestra el código necesario para obtener y almacenar periódicamente las coordenadas del usuario. En la siguiente sección, que trata sobre la implementación de las funcionalidades del mapa, se explicará el código usado para mostrar la ubicación del usuario en el mapa, en el que se usan métodos para tal fin además de usar las clases y métodos que se acaban de comentar.

```
public class Track implements LocationListener
{
    ...
    // Base de datos
    BD bd;

    private LocationManager lm;
    private String locationProvider;
```

```
public Track (Context context)
{
    // Obtención del objeto que maneja la base de datos
    bd = pfc.getBD();

    // Obtención de la instancia de LocationManager
    lm =
    (LocationManager)context.getSystemService(Context.LOCATION_SERVICE);

    // Definición de los criterios de selección del proveedor
    Criteria criteria = new Criteria ();
    criteria.setAccuracy(Criteria.ACCURACY_FINE);
    criteria.setCostAllowed(false);

    // Obtención del proveedor
    locationProvider = lm.getBestProvider(criteria, true);
    if (locationProvider != null)
    {
        // Programar actualizaciones de la ubicación cada 5 minutos
        // o cada 500 metros
        lm.requestLocationUpdates(locationProvider, 5*60000, 500, this);
    }
}

// Método que se invoca cuando se obtiene una nueva localizacion
@Override
public void onLocationChanged(Location location) {

    lm.removeUpdates(this);

    if (locationProvider != null)
    {
        lm.requestLocationUpdates(locationProvider, 5*60000, 500, this);

        // Obtención de la latitud y la longitud
        double latitude = location.getLatitude();
        double longitude = location.getLongitude();

        // Obtención y formateo de la fecha y hora en la que se han
        // recogido las coordenadas
        SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd");
        String date = sdf.format(Calendar.getInstance().getTime());
        sdf.applyPattern("HHmmss");
        String time = sdf.format(Calendar.getInstance().getTime());
```

```
            // Guardar las coordenadas, la fecha y la hora en la base de datos
            bd.open();
            bd.saveLocationPoint (latitude, longitude, date, time );
            bd.close();
        }
    }
        ...
```

Código 4.21: [Aplicación Android] Obtención de la localización.

Como se ve en el código, al método *setAccuraty* de la clase *Criteria* se le indica el parámetro "ACCURACY_FINE", el cual es necesario si se quiere poder obtener la localización mediante GPS. Otra opción hubiera sido "ACCURACY_COARSE", que indica una menor precisión en la ubicación que el parámetro anterior. Es por eso que "ACCURACY_FINE" sirve para localización mediante GPS o red, mientras que la cadena "ACCURACY_COARSE" sólo sirve para obtener la posición mediante redes.

Así pues, ya solo falta declarar en el fichero AndroidManifest.xml el permiso *android.permission.ACCESS_FINE_LOCATION* para que funcione todo el bloque de geolocalización implementado.

Mapas

Uno de los atractivos de desarrollar aplicaciones para esta plataforma es la fácil integración con mapas que permite. Google proporciona una librería externa específica para la representación de mapas en Android de forma nativa. Esta librería incluye el paquete `com.google.android.maps`, cuyas clases ofrecen entre otras opciones una variedad de controles para manejo del mapa.

Previamente a utilizar esta funcionalidad, es necesario que el desarrollador acepte los términos de uso del servicio de Google y obtenga una clave denominada "Maps API key", la cual deberá ser incluida en el código de la aplicación.

Las aplicaciones desarrolladas para Android necesitan tener un certificado asociado a las mismas para poder ejecutarse. Estos certificados son una forma de vincular la aplicación con su desarrollador, por lo que cada uno deberá tener su propio fichero de claves ".keystore" con el que crear certificados y poder publicar sus desarrollos. Si una aplicación está aún en desarrollo o en fase de pruebas, el SDK de Android proporciona un certificado especial para tal fin, de forma que las aplicaciones son firmadas en modo de pruebas.

Los pasos a realizar en la obtención de esta clave son los siguientes:

1. Si se está ejecutando la aplicación en modo de pruebas, se usará el fichero "debug.keystore" para generar una huella MD5 usando la herramienta *keytool.exe* [*Fig. 4.6*]. Esta herramienta se puede encontrar en la carpeta *bin* del JDK de Java. El comando necesario es:

```
keytool.exe -list -alias androiddebugkey -keystore
"C:\Temp\debug.keystore" -storepass android -keypass android
```

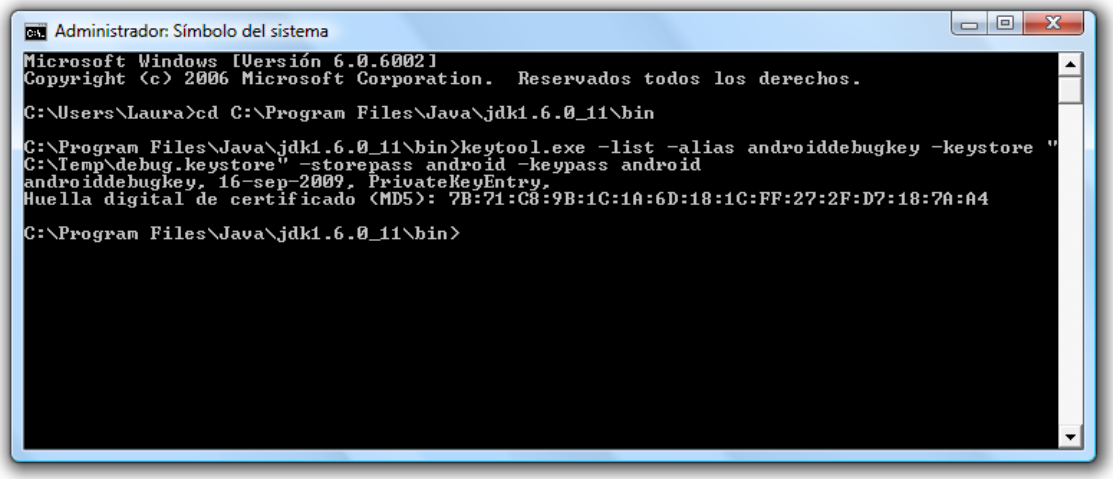

Figura 4.6: [Aplicación Android] Obtención de huella MD5.

2. Tras copiar la huella generada, para obtener la clave se siguen las instrucciones descritas en la página:

   http://code.google.com/intl/es-ES/android/maps-api-signup.html

   Básicamente, consiste en aceptar los términos del servicio y enviar mediante un campo de formulario la huella MD5 generada. Google devolverá entonces al usuario la clave para el uso de los mapas.

Aparte de obtener la *API key*, será necesario también modificar el fichero de declaraciones AndroidManifest.xml incluyendo la librería para el uso de mapas así como el permiso para el uso de internet.

```
<uses-library android:name="com.google.android.maps" />
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
```

Código 4.22: [Aplicación Android] AndroidManifest.xml: declaraciones para el uso de mapas.

Para crear la pantalla en las que se visualizan los mapas, se contruye por un lado el fichero .xml de *layout* y, por otro, la clase .java. A continuación se muestra el fichero .xml, en el que se define la vista *MapView*. Nótese que la clave obtenida anteriormente es incluida en este fichero en el atributo *apiKey* del elemento *MapView*.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <com.google.android.maps.MapView
        android:id="@+id/mapView"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:enabled="true"
        android:clickable="true"
        android:apiKey="0xpaxG2hE16efP_4qY_A8dJwLru7zVnw830Q1mQ"
        />

    <LinearLayout android:id="@+id/zoom"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        />

</RelativeLayout>
```

Código 4.23: [Aplicación Android] *Layout* de la pantalla *MapScreen*.

En cuanto al código .java, se ha creado una clase *MapScreen* que hereda de la clase *MapActivity*. Ésta cuenta con instancias de las clases necesarias para mostrar el mapa, que son *MapView* y *MapController*, como atributos. El código que muestra el mapa por pantalla es el siguiente:

```
// Creación del objeto mapView a partir del layout .xml
```

```
MapView mapView = (MapView) findViewById(R.id.mapView);
// Creación del objeto mapController
MapController  mapController = mapView.getController();
// Configuración de los controles que permiten hacer zoom
mapView.setBuiltInZoomControls(true);
// Definición del nivel de zoom que tendrá el mapa
mapController.setZoom(13);
```

Código 4.24: [Aplicación Android] Mostrar un mapa.

De momento, se podrá visualizar un mapa genérico, pero para mostrar las expediciones sobre dicho mapa, será necesario crear una capa nueva que contenga a las mismas y que será añadido al mapa. Esta capa está representada por la clase *TMMOverlay* que hereda de *Overlay*. Contiene dos *arrays* de objetos que representan las entregas y recogidas a mostrar en el mapa, las cuales serán pintadas mediante el método *draw(Canvas canvas, MapView mapView, boolean shadow)*. El algoritmo que sigue el método para pintar las expediciones es:

1. Se recorre el *array* de entregas.
2. Por cada entrega del array, se obtienen sus coordenadas geográficas (representadas mediante objetos *GeoPoint*) y se calcula la equivalencia a pixeles en la pantalla.
3. Se calcula si los pixeles en los que debe pintarse la entrega quedan dentro de la zona del mapa que es visible por pantalla.
4. Si la entrega debe ser mostrada, se obtiene el icono que la representará dependiendo del estado de la misma y se pinta dicho icono con el método *drawBitmap* del *Canvas*.
5. Se repiten los pasos anteriores para el *array* de recogidas.

En la siguiente sección de código se muestra parte del método *draw* anteriormente comentado:

```
public void draw(Canvas canvas, MapView mapView, boolean shadow)
{
    Projection projection = mapView.getProjection();
    Point screenPts = new Point();

    if (!shadow) {
        // Se recorre el array de entregas, que está definido fuera del método.
        for (int i = 0; i < itemsDelivs.size(); i++) {
```

```
            // Se obtienen las coordenadas de la entrega
            TMMOverlayItem item = itemsDelivs.get(i);
            GeoPoint geoPoint = item.getP();
            // Las coordenadas se convierten a pixeles
            projection.toPixels(geoPoint, screenPts);

            // Si la entrega debe ser mostrada en el área del mapa visible se
            // dibuja el icono correspondiente a su estado
            if (screenPts.x > -50 && screenPts.x < canvas.getWidth() + 50
                && screenPts.y > -50  && screenPts.y < canvas.getHeight() + 50)
            {
                // Icono gris si la entrega está pendiente
                if (item.getEstado().equals("1"))
                    canvas.drawBitmap(greyE,screenPts.x+8, screenPts.y-8, null);
                // Icono verde si la entrega está realizada sin incidencia
                else if (item.getEstado().equals("2"))
                    canvas.drawBitmap(greenE,screenPts.x+8,screenPts.y-8, null);
                // Icono verde si la entrega está realizada con incidencia
                else if (item.getEstado().equals("3"))
                    canvas.drawBitmap(greenE,screenPts.x+8,screenPts.y-8, null);
                // Icono rojo si la entrega está no realizada
                else if(item.getEstado().equals("4"))
                    canvas.drawBitmap(redE, screenPts.x+8, screenPts.y-8, null);
            }
        }
    ...
    }
}
```

Código 4.25: [Aplicación Android] Método *draw* de *TMMOverlay*.

Esta clase *TMMOverlay* también implementa la funcionalidad necesaria para mostrar información de las expediciones cuando son tocadas por el usuario en el mapa: el método *onTouchEvent(MotionEvent event, MapView mapView)* recoge el evento que es producido cuando el usuario pulsa sobre la pantalla, lo analiza para comprobar si ha sido una pulsación y, de ser así, muestra un diálogo con la información de la expedición.

Para diferenciar si el usuario ha pulsado sobre una expedición o estaba arrastrando el mapa, se obtienen las coordenadas de la pantalla en el momento de pulsar el mapa y las coordenadas en el momento en que el usuario levanta el dedo de la pantalla. Con ese par de coordenadas se calcula la distancia entre el primer punto y el segundo y si

la distancia es menor a un cierto parámetro, se considera que el usuario ha hecho una pulsación sobre un punto del mapa.

```
@Override
public boolean onTouchEvent(MotionEvent event, MapView mapView)
{
    switch (event.getAction())
    {
        // El usuario pone el dedo sobre la pantalla
        case MotionEvent.ACTION_DOWN:
            xDown = (int) event.getX(); // Coordenada x pulsada
            yDown = (int) event.getY(); // Coordenada y pulsada
            break;

        // El usuario levanta el dedo de la pantalla
        case MotionEvent.ACTION_UP:
            int xUp = (int) event.getX(); // Coordenada x al levantar el dedo
            int yUp = (int) event.getY(); // Coordenada y al levantar el dedo

            // Distancia entre los puntos de inicio y fin del movimiento
            double distance =
                Math.sqrt((xUp-xDown)*(xUp-xDown) + (yUp-yDown)*(yUp-yDown));

            // Si la distancia es pequeña, el evento es una pulsación
            if ( distance<25 )
            {
                // Mostrar el diálogo con información sobre la expedición
            }
            ...
```

Código 4.26: [Aplicación Android] Método onTouchEvent (I).

Una vez que se conoce que el movimiento realizado por el usuario sobre la pantalla ha sido una pulsación, lo siguiente es conocer la expedición sobre la que ha pulsado. Para ello, se obtienen las coordenadas del mapa sobre las que el usuario ha tocado la pantalla y se calcula la expedición situada geográficamente más cerca a dichas coordenadas. Esto es así porque al tocar la pantalla con los dedos no hay la precisión necesaria para pulsar exactamente sobre las coordenadas precisas de una expedición, y por tanto, rara vez se mostraría el diálogo de información.

La forma de calcular la expedición más próxima al punto de la pantalla tocado es usando el método estático *distanceBetween* de la clase *Location*, al cual se le pasan por

parámetro las coordenadas de los puntos entre los que se desea calcular la distancia. En el código de este proyecto, se recorren todas las expediciones de la capa para ir calculando la distancia de cada una de ellas al punto tocado, obteniendo la expedición más cercana como la que haya generado una menor distancia en los cálculos. A continuación se muestra el código de cálculo de la entrega con menor distancia al punto tocado. Posteriormente se realiza el mismo cálculo para las recogidas y se decide la expedición más cercana entre la entrega y la recogida resultante de cada cálculo.

```
// Distancia menor obtenida
float distanceE = 0;
// Índice de la entrega con menor distancia obtenida
int indexE = 0;

// Array donde se almacena la distancia calculada entre dos puntos
float[] results = new float[1];
// Se recorre el array de entregas
for (int i=0; i<itemsDelivs.size(); i++)
{
    TMMOverlayItem item = itemsDelivs.get(i);
    // Cálculo de la distancia entre el puntoTocado y una entrega
    Location.distanceBetween(pTouch.getLatitudeE6()/ 1E6,
        pTouch.getLongitudeE6()/ 1E6, item.getP().getLatitudeE6()/ 1E6,
        item.getP().getLongitudeE6() / 1E6, results);

    // Si la nueva distancia es menor que la anterior, se almacena
    if (i==0 || results[0]<distanceE) {
        distanceE = results[0];
        indexE = i;
    }
}
```

Código 4.27: [Aplicación Android] Cálculo de distancia entre dos coordenadas.

En el caso en que no hubiera ninguna expedición sobre el mapa, al tocar sobre él se mostrará la dirección sobre la que se ha pulsado. Para ello, se utiliza el servicio de geocodificación inversa de Google, el cual devuelve una dirección postal para un punto geográfico. De esta forma, gracias a las coordenadas del punto tocado en el mapa y el método *getFromLocation* de la clase *Geocoder* se puede obtener dicha dirección.

```
// Objeto que obtiene la dirección a partir de las coordenadas
Geocoder geoCoder = new Geocoder( context, Locale.getDefault());

// Lista de direcciones obtenidas para un punto geográfico
List<Address> addresses = geoCoder.getFromLocation(
p.getLatitudeE6() / 1E6,
p.getLongitudeE6() / 1E6, 1);

// Obtención de las direcciones asociadas a las coordenadas
String add = "";
if (addresses.size() > 0)
{
    for (int i=0; i<addresses.get(0).getMaxAddressLineIndex(); i++)
        add += addresses.get(0).getAddressLine(i) + "\n";
}
...
```

Código 4.28: [Aplicación Android] Geocodificación inversa.

Por último, antes de mostrar el diálogo informativo, se tiene en cuenta si la vista del mapa que se muestra al usuario es "street view" o no. En caso afirmativo, se añade un botón al diálogo que permite usar el servicio "street view" de Google para la dirección sobre la que se ha pulsado. El código necesario para ello es:

```
String uri = "google.streetview:cbll=" + pToStreetView.getLatitudeE6() / 1E6 +
"," + pToStreetView.getLongitudeE6() / 1E6 + "&cbp=1,45,,45,1.0&mz=5.0";

Intent intent = new Intent();
intent.setData(Uri.parse(uri));
intent.setAction("android.intent.action.VIEW");
context.startActivity(intent);
```

Código 4.29: [Aplicación Android] Visualizar en modo Street View.

De esta forma, se contruye la capa de las expediciones con toda la funcionalidad asociada a la misma. Queda por último añadirla al mapa para que sea visualizada por el usuario. Para ello, tras contruir la capa, se obtiene el listado de capas del mapa y se añade la nueva capa al mismo. Es importante eliminar posibles capas antiguamente añadidas que puedan estar desactualizadas.

```
// Capa de las expediciones
TmmOverlay capaExpediciones = new TmmOverlay(this, tmmOvItemArrayEntregas,
tmmOvItemArrayRecogidas, greenR, greenE, redR, redE, yellowR, greyE, greyR);
// Obtención de las capas asociadas al mapa
List<Overlay> mapOverlays = mapView.getOverlays();
// Borrar de capas antiguas
mapOverlays.clear();
// Añadir la capa con las expediciones
mapOverlays.add(capaExpediciones);
```

Código 4.30: [Aplicación Android] Añadir una capa al mapa.

Hasta el momento, cuando se inicia la pantalla del mapa podrá visualizarse el mapa y la capa de expediciones sobre el mismo. Por último, falta añadir una última capa sobre el mapa que representará la localización del usuario sobre el mismo. La creación de esta capa está en relación con la funcionalidad de obtención de las coordenadas geográficas explicada en la página 91. La clase *MapScreen* implementa el interfaz *LocationListener* y maneja objetos de las clases *LocationManager*, *Location* y *Criteria* para obtener actualizaciones de la ubicación del transportista. En este caso, el período de dichas actualizaciones es inferior al utilizado en la clase *Track*, la cual almacenaba las coordenadas. En este caso, las coordenadas no se almacenan, sino que son necesarias únicamente para mostrar la posición en el mapa.

La clase *FixedMyLocationOverlay* hereda de la clase *MyLocationOverlay*, la cual permite dibujar la posición del usuario en el mapa, además de mostrar una brújula sobre el mismo. Para activar estas capacidades que ofrece la capa, es necesario invocar los métodos *enableMyLocation()* y *enableCompass()* desde el método *onResume()* de la *activity*. De la misma forma, se desactivarán en el método *onPause* invocando *disableMyLocation()* y *disableCompass()*. La función de la clase *FixedMyLocationOverlay* es invocar *drawMyLocation* de la clase *MyLocationOverlay* de forma que la posición del dispositivo sobre el mapa se pinta automáticamente mostrando un icono azul parpadeante con un radio de incertidumbre alrededor. En caso de que esta representación fallara, esta clase se encarga de pintar por ella misma la ubicación del usuario, representándola gráficamente igual que lo hace la clase de la que hereda, pero sin parpadeo del icono.

Base de datos SQLite

La aplicación hace uso de una base de datos SQLite del dispositivo para almacenar todos aquellos datos que deben ser persistentes frente al apagado del terminal, pérdida de

batería, etc. En esta base de datos se almacenan los datos de validación del usuario, las entregas, recogidas, notificaciones y coordenadas geográficas recogidas por el dispositivo.

Las tres clases principales que se encargan del manejo de la base de datos son:

- Clase *BD*, cuyas instancias serán las que se manejarán por el resto de las clases del proyecto para realizar operaciones con la base de datos. Contiene todos los objetos y métodos necesarios para el manejo de la base de datos.
- Clase *DatabaseHelper*, hereda de *SQLiteOpenHelper* perteneciente al paquete *android.database.sqlite*. Expone todos los métodos necesarios para crear, borrar, ejecutar sentencias SQL y, en general, manejar otras tareas comunes de una base de datos.
- Clase *SQLiteOpenHelper*, perteneciente también al paquete *android.database.sqlite*, es una clase que sirve como ayuda para las operaciones de creación y control de versiones de la base de datos. Se encarga de obtener la base de datos o crearla si no existe y de actualizarla a una nueva versión si fuera necesario.

La clase *BD* tiene como atributos cadenas que definen todos los campos de todas las tablas, nombres y sentencias de creación de las tablas y nombre y versión de la base de datos. Además, contiene la clase estática *DatabaseHelper* con los métodos *onCreate* y *onUpgrade*, para crear y actualizar la base de datos, respectivamente:

```
private static class DatabaseHelper extends SQLiteOpenHelper
{
    // Constructor de la clase
    DatabaseHelper(Context context)
    {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    //Creación de las tablas
    @Override
    public void onCreate(SQLiteDatabase db)
    {
        db.execSQL(DATABASE_CREATE_USER_TABLE);
        db.execSQL(DATABASE_CREATE_DELIVERY_TABLE);
        db.execSQL(DATABASE_CREATE_COLLECTION_TABLE);
        db.execSQL(DATABASE_CREATE_MESSAGE_TABLE);
        db.execSQL(DATABASE_CREATE_LOCATION_POINTS_TABLE);
    }
```

```
    // Eliminación de las tablas existentes en la BD cuando se
    // actualiza la misma a otra versión
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
    {
        Log.w(TAG, "Upgrading database from version " + oldVersion + " to "
            + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS locationpoints;");
        db.execSQL("DROP TABLE IF EXISTS delivery;");
        db.execSQL("DROP TABLE IF EXISTS collection;");
        db.execSQL("DROP TABLE IF EXISTS message;");
        db.execSQL("DROP TABLE IF EXISTS user;");

        onCreate(db);
    }
}
```

Código 4.31: [Aplicación Android] Métodos de la clase *DatabaseHelper*

La operación de abrir la base de datos se realiza mediante el método *getWritableDatabase()* de la clase *DatabaseHelper*, el cual devuelve el objeto *SQLiteDatabase*. Para cerrar la base de datos se usa otro método de *DatabaseHelper* llamado *close()*. Sin embargo, para borrar la base de datos se hace a través de un objeto *File*, el cual se obtiene mediante la ruta donde se encuentra la base de datos.

Para escribir en la base de datos se utiliza la clase *ContentValues*, donde se especifican los campos y sus valores a almacenar. Después, dependiendo si la operación es de actualización de datos o de almacenamiento de datos nuevos, se usarán los métodos *update* e *insert*, respectivamente. Se muestra a continuación el método usado para guardar notificaciones llegadas al dispositivo desde el servidor, en el que se ilustra lo comentado anteriormente.

```
public boolean saveMessage( Message m)
{
    boolean saved = false;

    // Valores a guardar en la base de datos
    ContentValues args = new ContentValues();
    args.put(KEY_MESSAGE_CODE, m.getMessageCode());
    args.put(KEY_OPERATION, m.getOperation());
    args.put(KEY_STATE, m.getState());
```

```
    args.put(KEY_TEXT, m.getTxt());
    args.put(KEY_REPDATE, m.getRepDate());
    args.put(KEY_REPHOUR, m.getRepHour());
    args.put(KEY_SYNC, m.getSync());

    // Almacenamiento de los valores en el registro cuya clave
    // coincida con la del mensaje a almacenar
    int updateRows = mDb.update(DATABASE_TABLE_MESSAGE, args,
     KEY_MESSAGE_CODE + "="+ "'"+m.getMessageCode()+"'", null);

    // Si no se ha actualizado ninguna fila significa que el mensaje
    // es nuevo, y por tanto se inserta en la bd como tal
    if ( updateRows > 0)
        saved = true;
    else
    {
        if (mDb.insert(DATABASE_TABLE_MESSAGE, null , args) != -1)
        saved = true;
    }
    return saved;
}
```

Código 4.32: [Aplicación Android] Guardar información en la base de datos.

Para obtener datos de la base, se emplea el método *query*, al que se le pasa como parámetro el nombre de la tabla y opcionalmente, los campos a obtener y las cláusulas *WHERE*, *ORDER BY* y *HAVING*. A continuación se muestra un ejemplo de consulta en la que se obtienen todos los mensajes de la base de datos.

```
public Cursor fetchAllMessagesList()
{
    return mDb.query(DATABASE_TABLE_MESSAGE, new String[] {KEY_ROWID,
        KEY_TEXT, KEY_STATE}, null, null, null, null, KEY_STATE);
}
```

Código 4.33: [Aplicación Android] Hacer una consulta en la base de datos.

El último parámetro del método anterior es el campo "estado" del mensaje. Esto inplica que los mensajes serán devueltos ordenados ascendendemente según su estado, de forma que cuando se le muestra al usuario el listado de mensajes, aparecerán en la parte superior los no leídos, seguidos por los que han sido leídos y, cerrando la lista, los

que han sido borrados.

En cuanto al borrado de filas de la base de datos, se usa el método *delete* de la clase *SQLiteDatabase*, al que se le indica la tabla, y la claúsula "WHERE" para selecionar las filas a borrar. En el siguiente ejemplo de código, se muestra como eliminar un mensaje concreto de la base de datos, usando su código de mensaje para seleccionar la fila. En concreto, el método elimina de la base de datos los mensajes que ya han sido reportados al servidor y curyo estado es "borrado", para que solo desaparezcan del listado los que el usuario marcó como eliminados.

```
// Método que recibe un cursor con los mensajes ya sincronizados
public void deleteSentMessages (Cursor c)
{
    int cSize = c.getCount();

    for (int i=0; i<cSize; i++)
    {
        c.moveToPosition(i);
        // Si el estado es distinto de borrado, se marcan para que
        // no sean sincronizados en la siguiente sincronización
        String state = c.getString(2);
        if ( !state.equals("3") )
        {
            ContentValues args = new ContentValues();
            args.put(KEY_SYNC, "0");
            mDb.update(DATABASE_TABLE_MESSAGE, args,
                KEY_ROWID +"="+ "'"+c.getString(0)+"'", null);
        }
        // Si el usuario lo marc'o como borrado, se elimina de
        // la base de datos para no ser mostrado más
        else
            mDb.delete(DATABASE_TABLE_MESSAGE,
                KEY_ROWID +"="+ "'"+c.getString(0)+"'", null);
    }
}
```

Código 4.34: [Aplicación Android] Eliminación de filas de la base de datos.

Listados personalizados

La última parte que cabe destacar sobre la implementación del cliente Android es la creación de vistas personalizadas en las pantallas de listas de entregas, recogidas y notificaciones, cada una distinta de las demás. Las clases que representan estas pantallas heredan de la clase *ListActivity*, que facilita la representación de los datos en forma de lista. Todas ellas tienen un método *fillData()* encargado de obtener la información de la base de datos y asignar a la pantalla un adaptador para la lista. Es éste adaptador el que se ha personalizado para mostrar cada fila con un diseño concreto.

Se han creado tres clases que representan tres adaptadores distintos, cada uno para ser asignado a una pantalla de listado distinta. Estas clases mencionadas son *CustomAdapterDeliveries*, *CustomAdapterCollections* y *CustomAdapterMessages*, para las entregas, recogidas y mensajes, respectivamente. Estas clases contienen *arrays* de elementos a representar y métodos para obtener un elemento en concreto del *array* o el *ID* de un elemento según su posición. Lo interesante de estas clases es su método *getView*, el cual devuelve una vista concreta para cada elemento que se quiera añadir a la lista. Estas vistas son construidas mediante la clase *CustomAdapterView*, contenida en la clase anterior. Simplemente representan objetos *LinearLayout* a los que se le añaden los elementos deseados.

En el caso de las recogidas, para cada elemento de la lista se muestran dos líneas, una con un icono de estado, código de la recogida y cliente, y la segunda, con información sobre el campo "horario" de la misma. Dos clases distintas construyen la vista para un elemento recogida. La primera de ellas representa la primera fila y la segunda, incluye sobre la línea de horario que construye, la anterior vista con la primera fila.

La siguiente sección de código ilustra como construir listas de elementos personalizadas, tomando como ejemplo el listado de entregas.

```
public class CustomAdapterDeliveries extends BaseAdapter
{
        ...

    // Lista de objetos 'entrega'
    ArrayList<Delivery> dList;
        ...

    // Método que devuelve la vista asignada a cada elemento entrega
    @Override
```

```
    public View getView(int position, View contentView, ViewGroup parent)
    {
        Delivery d = dList.get(position);
        return new CustomAdapterView(this.context, d, position );
    }

    // Vista construida para cada fila de la lista de entregas
    class CustomAdapterView extends LinearLayout
    {
        public CustomAdapterView(Context context, Delivery d, int position) {
            super( context );

            // Icono circular del color que representa el estado de la entrega
            ImageView circleIW = new ImageView (context);
            circleIW.setImageResource(getCircle (d.getState(), position ));
            circleIW.setPadding(3, 2, 0, 2);

            // Texto que muestra el código de la entrega
            TextView codeTW = new TextView( context );
            codeTW.setText( d.getDelivCode() );
            codeTW.setTextColor(Color.BLACK);
            codeTW.setTextAppearance(context, android.R.style.
                TextAppearance_Medium );
            codeTW.setPadding(3, 0, 10, 0);

            // Texto que muestra el cliente de la entrega
            TextView clientTW = new TextView(context);
            clientTW.setText( d.getClient() );
            clientTW.setTextColor(Color.BLACK);
            clientTW.setTextAppearance(context, android.R.style.
                TextAppearance_Medium);
            clientTW.setEllipsize(TruncateAt.END);
            clientTW.setSingleLine();

            // Se añaden los elementos al layout
            addView(circleIW);
            addView(codeTW);
            addView(clientTW);
        }
    }
}
```

Código 4.35: [Aplicación Android] Adaptador de listas personalizado.

AndroidManifest.xml

El archivo AndroidManifest.xml finalmente queda de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1" android:versionName="1.0.0" package="uc3m.TMMCliente">

    <uses-permission android:name="android.permission.INTERNET">
    </uses-permission>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION">
    </uses-permission>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE">
    </uses-permission>
    <uses-permission android:name="android.permission.VIBRATE">
    </uses-permission>

    <uses-sdk android:minSdkVersion="7" />

    <application android:icon="@drawable/lorrygreen64x64"
        android:label="@string/app_name" android:debuggable="true" >

    <activity android:name=".screens.LoginScreen" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <activity android:name=".screens.MainMenuScreen"></activity>
    <activity android:name=".screens.DeliveriesListScreen"
            android:theme="@android:style/Theme.Light"></activity>
    <activity android:name=".screens.DeliveryDetailScreen"
            android:theme="@android:style/Theme.Light"></activity>

    <service android:name=".sync.SyncService">
        <intent-filter>
            <action android:name="uc3m.TMMCliente.SYNC_SERVICE" />
        </intent-filter>
    </service>

    <uses-library android:name="com.google.android.maps" />

    </application>
</manifest>
```

Código 4.36: [Aplicación Android] AndroidManifest.xml.

4.2.3. Perfil web de gestión

Este módulo del sistema es una herramienta web que permite monitorizar el estado de las expediciones así como la actividad de los transportistas. En la figura 4.7 se muestra el diagrama de navegación de este perfil de gestión.

Esta parte del sistema se ha desarrollado usando *servlets* y páginas JSP. Dentro de la estructura del directorio del proyecto encontramos la carpeta *src* contenedora de todas las clases .java y la carpeta *WebContent* que contiene, entre otros, los ficheros .jsp así como el archivo web.xml.

Ficheros .java

Los archivos .java están divididos en dos paquetes:

- **Paquete bd**: Contiene los beans de los elementos entrega, recogida, mensaje, localización y usuario. También contiene las clases *Conexion.java* y *Consulta.java* para realizar la conexión con la base de datos así como las operaciones sobre dicha base de datos, respectivamente. Por último, contiene una clase *Util.java* con métodos de formateo de horas y fechas.
- **Paquete servlets**: Contiene todos los *servlets* de la aplicación. Los *servlets* son clases java que heredan de *HttpServlet* y que implementan los métodos *doGet (HttpServletRequest request, HttpServletResponse response)* y *doPost (HttpServletRequest request, HttpServletResponse response)*. La llamada a cada método *doGet* siempre se redirige al método *doPost* donde se realizan las operaciones pertinentes según el caso, y en todos los casos la primera operación es comprobar si el usuario ha iniciado sesión de forma que se muestre la pantalla de *login* de no ser así.

  Los *servlets* implementados son los siguientes:

  - *CollectionData.java*: Obtiene toda la información de la base de datos de la recogida que el usuario seleccionó y la devuelve a la página *collectionData.jsp*.
  - *CollectionList. java*: Este *servlet* obtiene grupos de recogidas de la base de datos correspondientes a los campos del filtro. Si el usuario no usó los filtros, devuelve todas las recogidas almacenadas en la base de datos. Las recogidas son devueltas a la página *collectionList.jsp*.
  - *DeleteMessage.java*: Borra de la base de datos el mensaje indicado por el usuario. Después se pasa el control al *servlet MessageList.java*.

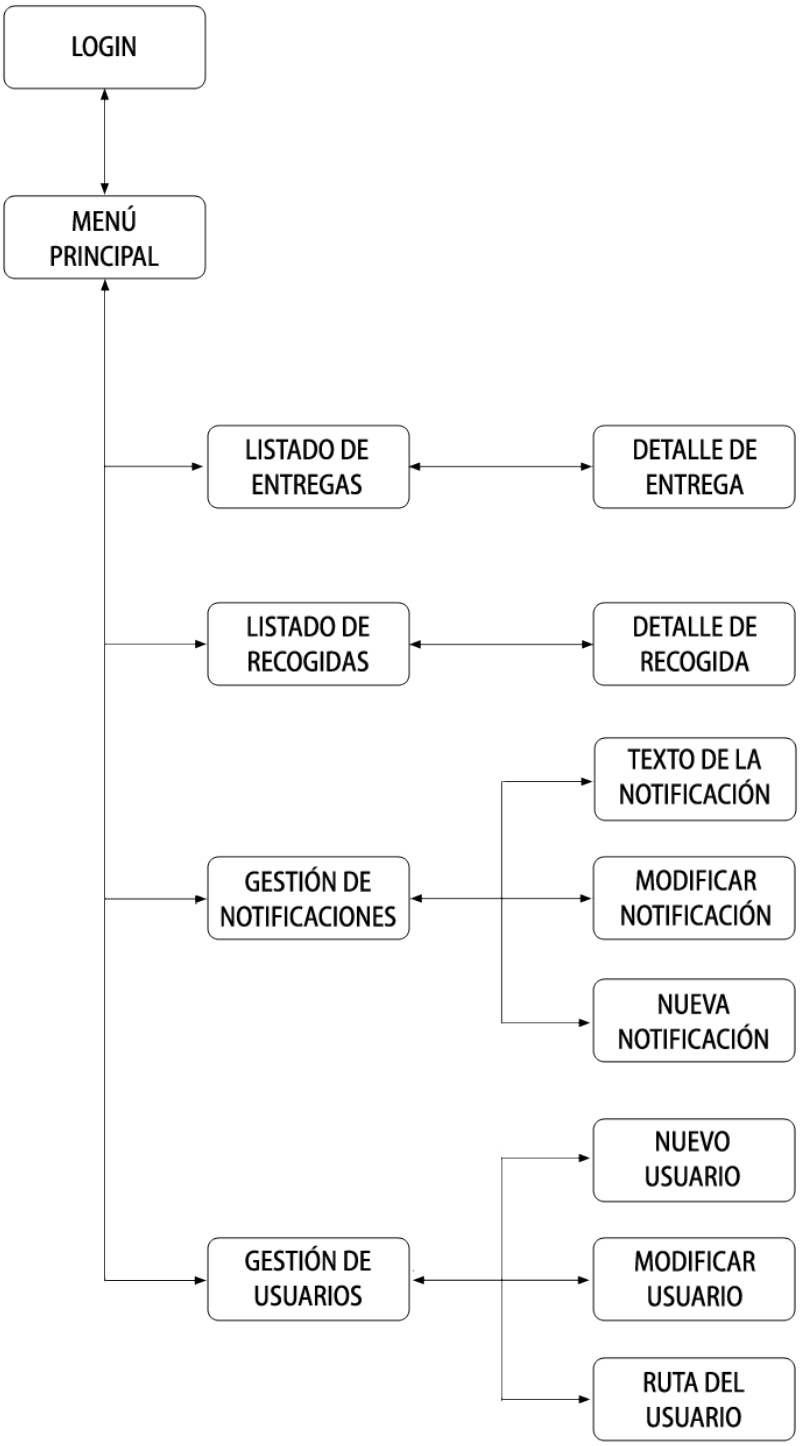

Figura 4.7: [Gestión web] Diagrama de navegación.

- *DeleteUser.java*: Borra de la base de datos el usuario indicado. Después se pasa el control al *servlet UserList.java*.
- *DeliveryData.java*: Obtiene toda la información de la base de datos de la entrega que el usuario seleccionó y la devuelve a la página *deliveryData.jsp*.
- *DeliveryList.java*: Este *servlet* obtiene grupos de entregas de la base de datos correspondientes a los campos del filtro. Si el usuario no usó los filtros, devuelve todas las entregas almacenadas en la base de datos. Dichas entregas son devueltas a la página *deliveryList.jsp*.
- *Login.java*: Este *servlet* valida en el sistema el usuario cuyo código y contraseña se introducen en la página de inicio de sesión. En caso de no estar completos alguno de esos dos campos se indica con un mensaje, al igual que si el usuario es incorrecto. En caso de introducción de un usuario válido se navega a la página del menú principal.
- *Logout.java*: Esta clase invalida la sesión y muestra la pantalla de *login*.
- *MessageDataForm.java*: Obtiene la información de un mensaje de la base de datos, tras lo que se muestra la página *messageForm.jsp*. En caso de haber algún error con el mensaje se vuelve a mostrar el listado de mensajes.
- *MessageList.java*: Este *servlet* obtiene mensajes de la base de datos correspondientes a los campos del filtro. Si el usuario no usó los filtros, devuelve todos los mensajes almacenados en la base de datos. Los mensajes son devueltos a la página *manageMessages.jsp*.
- *NewEditMessage.java*: Contiene el código necesario para las operaciones de alta y modificación de una notificación.
- *NewEditUser.java*: Contiene el código necesario para llevar a cabo las operaciones de creación y modificación de un usuario.
- *UserDataForm.java*: Obtiene la información de un usuario de la base de datos, tras lo que se muestra la página *userForm.jsp*. En caso de haber algún error con el usuario se vuelve a mostrar el listado de usuarios.
- *UserList.java*: Este *servlet* obtiene usuarios de la base de datos correspondientes a los campos del filtro. Si el usuario no usó los filtros, devuelve todos los usuarios almacenados en la base de datos. Los usuarios son devueltos a la página *manageUsers.jsp*.
- *UserLocPoints.java*: Devuelve en un .xml las coordenadas, entregas no pendientes y recogidas no pendientes de un usuario seleccionado, de forma que esta información será mostrada en el mapa.

Ficheros .jsp

En cuanto a los ficheros .jsp, son páginas que contienen formularios, tablas, hojas de estilo y funciones *javascript* para controlar algunos comportamientos de la interfaz. Cabe destacar, sin embargo, la página encargada de mostrar los mapas de rutas de usuarios.

Para la funcionalidad de mostrar los mapas de Google con las rutas y expediciones de cada usuario ha sido necesario utilizar en las páginas .jsp los lenguajes *javacript* y AJAX. Por otra parte, cuando es necesario mostrar un número elevado de coordenadas en un mapa a la hora de pintar rutas se requiere bastante cantidad de memoria y, a menudo, puede llevar mucho tiempo dibujar dichas rutas. Para evitar esta situación, se hace necesario usar polilíneas codificadas mediante un formato comprimido de caractéres ASCII. Google proporciona una explicación sobre los pasos del algoritmo de codificación a emplear [40], pero no proporciona ninguna implementación de este algoritmo, la cual no es sencilla ni trivial. Dado que el objetivo del proyecto difiere bastante de implementar algoritmos que por otra parte ya pueden encontrarse implementados, se ha integrado en el código de la herramienta el fichero *PolylineEncoder* realizado por Mark McLure [41].

En el código HTML hay dos líneas en relación a la visualización del mapa. La primera es la llamada a los métodos *initialize()* y *GUnload()* en los eventos *onload* y *onunload* del elemento *body*, respectivamente. Estos métodos sirven para inicializar el mapa y liberar recursos. La segunda línea consiste simplemente en definir un elemento *div* que contenga el mapa.

El código más extenso y complicado relacionado con mostrar el mapa es el implementado en las funciones *javascript*. En primer lugar, es necesario incluir en la cabecera del documento los siguientes *scripts*:

```
<script src="http://maps.google.com/maps?file=api&amp;v=2.173&amp;key=
    ABQIAAAAD5zQIQyoeG0ckgBMSCUOwhTwM0brOpm-
    All5BF6PoMKBxR4WERRyW1AHNmsIgcvemk727e_UMiVsRA&sensor=true" type="text/
    javascript">
</script>

<script src="PolylineEncoder.js" type="text/javascript"></script>
```

Código 4.37: [Gestión web] *Scripts* de *userMap.jsp*.

El primero de ellos es necesario para usar el API de *Javascript* de Google Maps. Se le indican, entre otros parámetros, la clave obtenida mediante registro y el valor del parámetro "sensor". Éste será verdadero o falso dependiendo de si la aplicación utiliza algún dispositivo para obtener la posición del usuario, en este caso, un receptor GPS. El segundo *script* es para incluir el algoritmo de codificación desarrollado por Mark McLure.

Los métodos *javascript* principales involucrados en la visualización del mapa y su contenido son tres:

- Función *getIcon (state, type)*: Construye y devuelve una variable de tipo *GIcon()* utilizando los valores que recibe como parámetro para asignar a dicha variable los valores correspondientes. La variable *type* marca si el icono será de una entrega o una recogida, mientras que la variable *state* define el color del icono dependiendo del estado del la expedición.
- Función *addTag (point, item, type)*: Esta función contruye los marcadores mostrados en el mapa. Se entiende por "marcador" las expediciones que se representan en el mapa. En este proyecto, un marcador está compuesto de las coordenadas sobre las que se representa el mismo, un icono y una acción. Las coordenadas son representadas mediante un objeto *GPoint*, el icono se obtiene invocando al método anteriormente explicado, y la acción se ejecuta sobre el evento de pulsar con el ratón sobre el marcador, de forma que aparece un cuadro sobre el mapa con información específica sobre la expedición pulsada.
- Función *initialize()*: Este función contiene el código AJAX para recarga dinámica de información en la página. Obtiene, a través del servlet *UserLocPoints*, tanto las coordenadas enviadas de forma periódica por el usuario como sus entregas y recogidas no pendientes. La información obtenida por el *servlet* es devuelta en un fichero .xml con el siguiente formato:

```
<?xml version="1.0" encoding="UTF-8"?>

    <xml>
        <track>
            <trkpt>
                <trklatitude>40.418023</trklatitude>
                <trklongitude>-3.69278</trklongitude>
            </trkpt>
            <trkpt>
                <trklatitude>40.412362</trklatitude>
                <trklongitude>-3.665956</trklongitude>
            </trkpt>
            <trkpt>
                <trklatitude>40.406928</trklatitude>
```

```
            <trklongitude>−3.673339</trklongitude>
        </trkpt>
        ...
    </track>

    <delivMarkers>
        <delivMarker>
            <mCod>100012</mCod>
            <mState>Realizada con incidencia</mState>
            <mStateInt>2</mStateInt>
            <mClient>Buceo Espeleologia y Montaña, SL</mClient>
            <mAddress>Avenida del Padre Piquer 58, Madrid</mAddress>
            <mDateTime>11−09−2010 − 15:32</mDateTime>
            <mLatitude>40.390897</mLatitude>
            <mLongitude>−3.766959</mLongitude>
        </delivMarker>
        ...
    </delivMarkers>

    <collectMarkers>
        <collectMarker>
        <mCod>100002</mCod>
        <mState>Realizada</mState>
        <mStateInt>2</mStateInt>
        <mClient>CSN</mClient>
        <mAddress>C/Pedro Justo Dorado Dellmans 11, Madrid</mAddress>
        <mDateTime>11−09−2010 − 16:25</mDateTime>
        <mLatitude>40.448057</mLatitude>
        <mLongitude>−3.714044</mLongitude>
    </collectMarker>
    ...
</collectMarkers>
```

Código 4.38: [Gestión web] Información devuelta por el *servlet UserLocPoints*.

A partir de las coordenadas, se crean variables de tipo *GLatLng* que se almacenan en un array. Este array es el que se le pasa al método *dpEncodeToGPolyline* del algoritmo de Mark McLure para obtener las poliníneas codificadas que representan las rutas de los usuarios. Estas polilíneas son añadidas al mapa mediante el método *addOverlay* del objeto *GMap2*.

Tanto las entregas como las recogidas se usan para crear marcadores sobre el mapa que las representen. Se hace con las funciones comentados anteriormente. Los marcadores se añaden al mapa de la misma forma que las polilíneas.

Esta función también añade los controles para manejar el mapa y lo centra en la posición y nivel adecuados.

Por su nivel de interés, se incluye a continuación parte del código que forma la función:

```
function initialize()
{   ...

    // Declaración de variables que serán usadas posteriormente
        ...
    var map = new GMap2(document.getElementById("map_canvas"));
        ...

    // Petición de las localizaciones y expediciones del usuario
    oXML.open("GET", "servlets/UserLocPoints?userSelected="+userSelected)
        ;
    oXML.onreadystatechange = function() {
        if(oXML.readyState == 4) {
            var xml = oXML.responseXML;
            try {
                // Variable para codificar los puntos de la ruta
                polylineEncoder = new PolylineEncoder();
            }catch(err) {  alert (err.description); }

            // Array que almacenará los puntos que forman la ruta
            var points = new Array(0);

            // Almacenado de los puntos que forman la ruta en el array
            // points para su codificación
            for ( var i = 0; i < xml.getElementsByTagName("trkpt").length;
                 i++)
            {
                var item = xml.getElementsByTagName("trkpt")[i];
                    lat = item.getElementsByTagName("trklatitude")[0].
                        firstChild.data;
                lon = item.getElementsByTagName("trklongitude")[0].
                    firstChild.data;

                points[i] = new GLatLng(lat,lon);
            }

            // Obtención de la polilínea codificada que forma la ruta
            // a partir del array de puntos correspondientes a las
            // coordenadas del usuario
          polyline = polylineEncoder.dpEncodeToGPolyline(points);
```

```
            // Adición de la polilínea al mapa
            map.addOverlay(polyline);

            //Centrado del mapa
            if (lat!=null)
                map.setCenter(new GLatLng(lat, lon), 12);
            else
                map.setCenter(new GLatLng(40.420088, -3.088000), 6);

            // Creación de marcadores para cada entrega
            for ( var j = 0; j < xml.getElementsByTagName("delivMarker").
                length; j++)
            {
                var deliv = xml.getElementsByTagName("delivMarker")[j];

                var mLat = deliv.getElementsByTagName("mLatitude")[0].
                    firstChild.data;
                var mLon = deliv.getElementsByTagName("mLongitude")[0].
                    firstChild.data;
                var point = new GLatLng (mLat, mLon);
                var marker = addtag(point, deliv, "d");

                //Adición del marcador al mapa
              map.addOverlay(marker);
            }
            ... // Creación de los marcadores de recogidas
        }
    };

    oXML.send('');

    // Centrado del mapa
    if (lat!=null)
        map.setCenter(new GLatLng(lat, lon), 9);
    else
        map.setCenter(new GLatLng(40.420088, -3.088000), 5);

    // Adición de los controles del mapa
    map.addControl(new GLargeMapControl());
    map.addControl(new GMapTypeControl());

    ...
```

Código 4.39: [Gestión web] Función *initialize()*

Fichero web.xml

En este fichero, contenido en la subdirectorio WEB-INF de la carpeta WebContent, se definen los *servlets* de la aplicación de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
    com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" xsi:
    schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    app_2_5.xsd" id="WebApp_ID" version="2.5">

  <display-name>TMMManagerProfile</display-name>

  <welcome-file-list>
    <welcome-file>mainMenu.jsp</welcome-file>
  </welcome-file-list>

  <servlet>
    <description></description>
    <display-name>Login</display-name>
    <servlet-name>Login</servlet-name>
    <servlet-class>servlets.Login</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Login</servlet-name>
    <url-pattern>/servlets/Login</url-pattern>
  </servlet-mapping>

  <servlet>
    <description></description>
    <display-name>UserList</display-name>
    <servlet-name>UserList</servlet-name>
    <servlet-class>servlets.UserList</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>UserList</servlet-name>
    <url-pattern>/servlets/UserList</url-pattern>
  </servlet-mapping>

  ...
```

Código 4.40: [Gestión web] Fichero web.xml.

4.2.4. Perfil web de introducción de expediciones

La aplicación de introducción de expediciones en el sistema es muy sencilla, tal y como se puede apreciar en el diagrama de navegación de la figura 4.8.

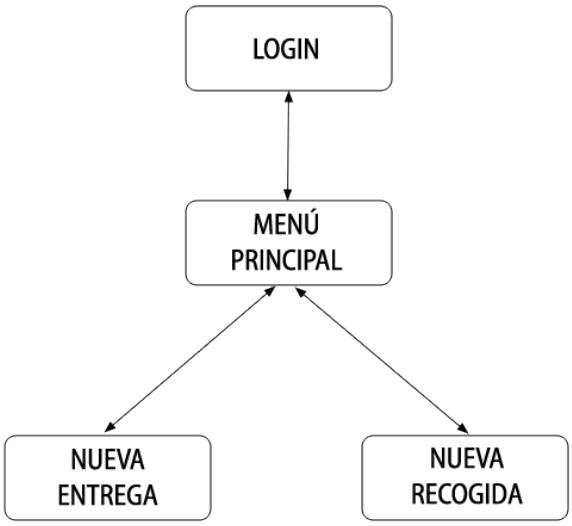

Figura 4.8: [Introducción de expediciones] Diagrama de navegación.

Al igual que en el módulo anterior, éste es una aplicación basada en *servlets* y páginas JSP. Las clases java están divididas en los paquetes *bd* y *servlets*. Las clases pertenecientes al paquete *bd* son los *beans* de entrega, recogida y usuario, las clases de conexión y consulta a la base de datos y una última clase de utilidades para el formateo de fechas y horas. En el paquete *servlets* hay cuatro *servlets*:

- Clase *Login*: Comprueba si el usuario y contraseña son válidos. De ser así, se almacena este valor en el entorno de sesión y se fija un periodo de expiración de la misma de media hora. Seguidamente se navega al menú principal.
- Clase *GetDataForm*: Este *servlet* sirve para navegar desde el menú principal a los formularios de expediciones según corresponda a lo seleccionado en dicho menú.
- Clase *NewDelivery*: En esta clase se efectúa el guardado de la nueva entrega introducida. Si se produce algún error se notifica al usuario en el mismo formulario de introducción de datos. En caso contrario se regresa al menú principal.

- Clase *NewCollection*: En esta clase se efectúa el guardado de la nueva recogida introducida. Si se produce algún error se notifica al usuario en el mismo formulario de introducción de datos. En caso contrario se regresa al menú principal.

La primera operación realizada siempre por cada uno de los *servlets* es comprobar si la sesión del usuario está activa, para regresar a la pantalla de inicio de sesión de no ser así.

Para esta parte se ha seguido una estética igual a la de la anterior aplicación web, usando para ello los mismos estilos en las páginas .jsp. Hay únicamente cuatro páginas .jsp, correspondientes a cada una de las pantallas mostradas en el diagrama de navegación:

- Página *login.jsp*: Es la pantalla de inicio de sesión.
- Página *mainMenu.jsp*: Pantalla del menú principal.
- Página *deliveryEntry.jsp*: Esta página es el formulario de recogida de datos para una nueva entrega.
- Página *collectionEntry.jsp*: Formulario para la creación de una nueva recogida.

Todas las páginas se componen de elementos sencillos como títulos, tablas o formularios. Cabe destacar en los formularios de creación de expediciones la funcionalidad de la obtención de las coordenadas geográficas de esas expediciones automáticamente y de forma transparente al usuario a partir de la dirección introducida por éste. Gracias a estas coordenadas, las expediciones podrán ser situadas en los mapas. Para ello, se ha implementado una función *javascript* que es la que se encarga de realizar esta operación.

En primer lugar, es necesario incluir en la cabecera del documento el *script* que permite usar el API de *Javascript* de Google Maps. Es el mismo *script* que el usado en la aplicación de gestión web. Por otra parte, se han incluido en el formulario dos campos ocultos correspondientes a la latitud y la longitud de la expedición. De esta forma, la función *javascript getLocationPoint()* rellenará estos campos para que sean almacenados en la base de datos junto con el resto de información. Esta función es llamada en el evento "onblur" de los campos "Dirección", "Población" y "País". En primer lugar la función comprueba si estos tres campos están rellenos, y en tal caso, utiliza una variable *GClientGeocoder()* para obtener un punto geográfico a partir de la dirección compuesta por esos tres campos.

Se incluye a continuación la función *getLocationPoint()* usada:

```
function getLocationPoint()
{
    // Campos del formulario que forman la dirección completa
    direccion = document.getElementById("direccion");
    poblacion = document.getElementById("poblacion");
    pais = document.getElementById("pais");

    // Comprobación de que todos ellos están rellenos
    if (direccion.value != "" &&
        poblacion.value != "" &&
        pais.value != "")
    {
        // Cadena con la dirección completa
        var address = direccion.value + ", " +
        poblacion.value + ", "+
        pais.value ;

        // Variable del API de Google para la geocodificación
        var geocoder = new GClientGeocoder();

        // Geocodificación
        geocoder.getLatLng( address,
            function(point) {
                if (!point) {}
                else
                {
                    // Asignación de la latitud y longitud a los campos del
                        formulario
                    document.getElementById("latitud").setAttribute("value",
                        point.lat());
                    document.getElementById("longitud").setAttribute("value",
                        point.lng());
                }
            }
        );
    }
}
```

Código 4.41: [Introducción de expediciones] Función *getLocationPoint()*

Capítulo 5

CONCLUSIONES Y LÍNEAS FUTURAS

En este capítulo se exponen las conclusiones obtenidas tras la realización del proyecto, haciendo un repaso global del mismo y comparando los resultados obtenidos con los objetivos marcados inicialmente.

5.1. Conclusiones

El estudio sobre las tecnologías a emplear, análisis y diseño de la herramienta, implementación en código y período de pruebas han sido las principales fases de este proyecto. Tras la conclusión de todas estas etapas que han constituido el desarrollo del proyecto, es momento de hacer balances sobre los resultados obtenidos. El objetivo principal marcado en el inicio de este proyecto fue construir una herramienta que permitiera agilizar las actividades de gestión de expediciones de una empresa de mercancías. Dicho objetivo puede decirse que se ha cumplido satisfactoriamente. A continuación se hace un repaso del estado de los componentes que forman la herramienta:

- Desarrollo de un cliente móvil sobre Android para uso de los transportistas:
  Se ha conseguido implementar una aplicación que consigue agilizar la actividad diaria de los transportistas. El interfaz mostrado al usuario es atractivo y la navegación, sencilla. La integración de otras aplicaciones en la herramienta la dotan de una mayor utilidad. El principal atractivo de este módulo puede decirse que son los mapas que puede visualizar el usuario, con opciones disponibles como la visualización en modo "street view", donde pueden verse imágenes reales de localizaciones. Cabe destacar sobre este aspecto el atractivo que añade el modo brújula integrado

en la vista "street view".

- Desarrollo de una aplicación web para la supervisión y gestión de la actividad de los transportistas. Se ha conseguido implementar una herramienta web de diseño atractivo y navegación muy intuitiva. Cumple con todas las funcionalidades que se esperaba: consulta de expediciones y creación, modificación y borrado de mensajes y usuarios. A la funcionalidad básica de la herramienta se le incluyeron mapas donde seguir de forma visual la actividad de los transportistas.

- Interfaz de introducción de entregas y recogidas. En este caso no solo se ha conseguido desarrollar una herramienta para introducir expediciones en el sistema, sino que se le ha añadido una funcionalidad no contemplada inicialmente. Dicha funcionalidad consiste en la obtención automática y transparente de las coordenadas geográficas de una dirección que introduce el usuario. En un principio se iban a mostrar campos de latitud y longitud en el formulario de recogida de datos, pero gracias al servicio de Google de geocodificación, se ha mejorado esta funcionalidad.

La tecnología más innovadora o moderna del proyecto es la referente a la plataforma Android. Personalmente como desarrolladora del proyecto he disfrutado aprendiendo y programando en esta nueva plataforma. Aunque en un principio el avance es lento, al final tras comprender la filosofía de la plataforma y todo lo que la rodea, se pone de manifiesto la potencia que tiene dicho sistema con mecanismos tan novedosos como los *intents* o la fácil integración de otros servicios. El balance referente al uso de esta plataforma es muy positivo. Las características que llevan a esta conclusión son las siguientes:

- Android es *open source*, lo cual significa que cualquier persona puede contribuir a mejorar el sistema.

- Tiene el respaldo de las empresas más importantes, tanto a nivel de fabricantes, como operadoras.

- El crecimiento de la plataforma es constante y se puede decir que está en pleno auge.

- Su aplicación "Android Market" pone a disposición de los usuarios una amplia gama de aplicaciones de todo tipo.

- Google ha realizado un buen trabajo en cuanto a documentar la plataforma, centralizando toda la información útil para los desarrolladores en su página web *http://developer.android.com*. A parte, actualmente la comunidad Android en internet ha crecido mucho, de forma que es posible encontrar *blogs*, foros y webs sobre desarrollo donde obtener información igualmente muy útil.

- El hecho de que el lenguaje usado para desarrollar sobre esta plataforma sea Java con archivos xml para los interfaces de usuario facilita mucho la programación, dado que ya se tiene este conocimiento de forma previa a iniciarse en Android.

- Las herramientas de desarrollo facilitan mucho el trabajo del programador. Un ejemplo es el *plugin* para el IDE Eclipse y el emulador donde probar el código desarrollado. Un punto a favor de dicho emulador es que permite probar código modificado sin necesidad de reiniciarlo y esperar demasiado.

- La posibilidad de almacenar los datos de forma persistente en una base de datos SQLite es un punto a favor del sistema también. Las funciones de almacenamiento de información sobre esta base de datos son mucho más sencillas de implementar que sobre otras plataformas como por ejemplo Blackberry, donde para las versiones anteriores a la 5.0 solo cabía la posibilidad de utilizar las clases *RecordStore* y *PersistentStore*.

- La construcción de los interfaces gráficos se ve facilitada gracias al entorno gráfico que se proporciona para ello, además de su sistema de archivos .xml que se usan para tal fin.

- Cabe destacar, por supuesto, lo fácil que es integrar una aplicación con los servicios que proporciona Google, lo cual dota de mayor potencia a las aplicaciones desarrolladas para este sistema.

- La confirmación por parte del usuario que se impone para utilizar ciertas capacidades del teléfono por parte de aplicaciones de terceros proporciona un nivel de seguridad a la plataforma carente en otros sistemas.

Críticas y dificultades encontradas

Por la imposibilidad de haber tenido una dedicación a tiempo completo al proyecto, debido a motivos laborales, el desarrollo del mismo se ha extendido bastante en el tiempo. Este hecho ha tenido consecuencias tanto positivas como negativas. Por ejemplo, al inicio de comenzar con la codificación del cliente Android, superar las dificultades que iban surgiendo fue una tarea que en ocasiones se alargaba demasiado debido a la falta de documentación y ejemplos que podían encontrarse por internet. Con el paso de los meses, cada vez surgieron más páginas webs, *blogs* y foros sobre la plataforma Android en las que encontrar información útil. Sin embargo, una de las consecuencias no tan positivas fue el vivir en primera persona el problema de la fragmentación en Android, talón de Aquiles de la plataforma condicionada por el hardware, fabricantes, operadoras o la propia Google.

Cuando se habla de fragmentación del sistema Android, se hace referencia al amplio número de versiones distintas de la plataforma que coexisten a la vez en el mercado. Actualmente, el mayor porcentaje de dispositivos Android portan una de las siguientes cuatro versiones: 1.5, 1.6, 2.1 y 2.2, quedando un 0,2 % de dispositivos con versiones que Google califica de obsoletas. Han pasado ya cuatro meses desde que Google lanzara la versión 2.2 *Froyo* de la plataforma, sin embargo, un mínimo porcentaje de usuarios utiliza actualmente dicha versión y, sorprendentemente, aún un 35 % de los terminales Android se encuentran en las versiones 1.5 y 1.6, según datos oficiales de Google a fecha del dos de Agosto de este año [39].

En la figura 5.1 se muestra gráficamente la evolución de esta fragmentación durante los últimos seis meses. Estos datos son obtenidos por Google mediante la comprobación del sistema que portan los dispositivos que acceden al Android Market durante dos semanas.

La fragmentación provoca que los usuarios dispongamos de una plataforma que tiene desventajas serias con respecto a la de las versiones más modernas. Muchos usuarios ven cómo su terminal queda totalmente desactualizado a pesar de no haber pasado ni un año desde que lo adquirieron. En cuanto a los desarrolladores de aplicaciones, el problema consiste en que si programan para la última versión de Android, pueden dejar fuera a un gran porcentaje de los usuarios del sistema operativo. Es lo que sucedió, por ejemplo, con la aplicación de *Twitter* cuando se estrenó para Android, que sólo funcionaba con la versión 2.1. De esta forma, los desarrolladores se ven obligados a mantener una compatibilidad entre las diversas plataformas, o bien desarrollar varias versiones de sus aplicaciones para adaptarlas a los distintos sistemas operativos del mercado.

Como he comentado, me he visto afectada por el problema tanto a nivel de usuario al poseer un terminal Motorola Milestone, como a nivel de desarrollador al haber implementado la herramienta *TMM*. Al poco de adquirir el dispositivo con versión 2.0, apareció la versión 2.1 del sistema con características y funcionalidades nuevas de las que mi terminal no disponía. En cuanto al desarrollo de la herramienta *TMM*, he tenido que ir cambiando de plataforma sucesivas veces durante el desarrollo, modificando código en algunos casos y comprobando cómo para unas versiones la herramienta funcionaba bien y para otras no.

Referente a este tema, cabe añadir sin embargo que parece que Google está trabajando en eliminar este problema. Para ello, Google extraeráa del núcleo de Android algunas características y mejoras de usabilidad, además de sus aplicaciones propietarias, que estarían disponibles para descargar desde el Android Market. Así los usuarios no tendrían la necesidad de esperar a que los fabricantes de terminales u operadoras se lancen las actualizaciones compatibles con los distintos tipos de móviles. Además, parece ser que

Figura 5.1: Evolución de la fragmentación en el sistema Android.

también está pensado reducir el número de actualizaciones que se hacen al sistema operativo para así ayudar a que los operadores y los fabricantes tengan tiempo de adoptar actualizaciones.

En cuanto a las dificultades encontradas en la codificación del proyecto, la implementación de ciertas partes fue especialmente difícil.

En primer lugar, a la hora de utilizar la librería *kSoap2* para la comunicación del cliente móvil con el servidor aparecieron dificultades cuya solución se encontró a través del método prueba y error, debido a la falta de documentación y ejemplos prácticos que podían encontrarse por internet en aquel entonces. Actualmente, ya existen tutoriales y ejemplos sobre como usar esta librería.

La dificultad más importante que hubo que salvar en el desarrollo del proyecto está relacionada con la implementación de mapas dinámicos de Google para la interfaz web. Google proporciona un API muy completo con ejemplos muy bien documentados sobre como usar dicho API para la creación de mapas dinámicos. Sin embargo, el mostrar un número elevado de coordenadas en un mapa a la hora de pintar rutas requiere bastante cantidad de memoria y, a menudo, puede llevar mucho tiempo dibujar dichas rutas. Para evitar esta situación, se hace necesario usar polilíneas codificadas mediante un formato comprimido de caractéres ASCII. Google proporciona una explicación sobre los pasos del algoritmo de codificación a emplear [40], pero no proporciona ninguna implementación de este algoritmo, la cual no es sencilla ni trivial. La aparición de esta dificultad supuso el planteamiento de cambiar los mapas dinámicos por mapas estáticos, pero finalmente se integró en el proyecto el código *PolylineEncoder* realizado por Mark McLure [41]. Este fichero implementa en javascript el algoritmo de codificación descrito por Google. Aún así, no fue sencilla la implementación de esta funcionalidad.

5.2. Trabajos futuros

Como líneas de trabajo futuras, se propone ampliar y mejorar la funcionalidad de la herramienta con las siguientes sugerencias:

- Debido a que el módulo del cliente móvil ha sido desarrollado sobre la plataforma Android, su usabilidad está limitada por tanto a los terminales con dicho sistema

operativo. La implementación de un interfaz web móvil transformaría la aplicación en multiplataforma, de forma que podría ser utilizada desde terminales de otro tipo y con otro sistema operativo.

- Una mejora que sería muy útil para el transportista sería poder visualizar las expediciones ordenadas secuencialmente de forma que el tiempo empleado en completar su hoja de ruta sea el menor posible. Esto debería implementarse mediante algún tipo de algoritmo a nivel servidor, el cual enviaría al dispositivo móvil las expediciones ya ordenadas. Esto tendría además la ventaja añadida de un ahorro considerable de combustible por el vehículo utilizado, reduciendo así costes.

  Analizando esta necesidad, podemos comprobar que nos encontramos ante el famoso *Problema del viajante* o *TSP: Travelling Salesman Problem*. Éste se basa en, dadas N ciudades, encontrar una ruta que comenzando y terminando en una ciudad concreta pase una sola vez por cada una de ellas y minimice la distancia recorrida por el viajante. En nuestro caso, las ciudades serían sustituidas por expediciones y el punto de comienzo y final sería la oficina o central. La respuesta al problema es conocida, es decir se conoce la forma de resolverlo, pero sólo en teoría, en la práctica la solución no es aplicable debido al tiempo que computacionalmente se precisa para obtener su resultado.

  El TSP constituye la situación general y de partida para formular otros problemas combinatorios más complejos, aunque más prácticos, como por ejemplo los problemas de rutas de vehículos (*Vehicle Routing Problem - VRP*) [44]. En concreto, el problema llamado VRPPD (*VRP with Pickup and Delivery*) es el que más se ajusta a la funcionalidad de este proyecto. Los algoritmos clásicos no son capaces de resolver el problema general, debido a la explosión combinatoria de las posibles soluciones. Por ello, a su solución se aplican distintas técnicas computacionales: heurísticas evolutivas, redes de Hopfield, etc.

  Podría intentar implementarse algún tipo de solución siguiendo estas técnicas de forma que el funcionamiento de la herramienta se viera complementado.

- La funcionalidad de la herramienta se podría mejorar en la parte web de la misma, añadiendo la posibilidad de envío de mensajes a grupos de transportistas o incluso a todo el conjunto de ellos. Actualmente, un nuevo mensaje es asignado a un único usuario.

- La integración de otros módulos y aplicaciones Android en la herramienta complementarían las funciones que ya ofrece y aumentaría la utilidad de la misma. Incluso podrían definirse varios perfiles móviles según el tipo de reparto que se vaya a realizar. Por ejemplo, mostrar aplicaciones integradas diferentes para un repartidor a pie y para un repartidor con vehículo motorizado.

- También sería interesante la integración con otros sistemas. En este proyecto se ha usado una base de datos MySQL, servicios web en Java y aplicaciones web mediante

servlets y JSPs, además del desarrollo móvil sobre la plataforma Android. Teniendo en cuenta que la aplicación está destinada a un ámbito empresarial, sería interesante versionar la parte servidor y de base de datos para sistemas como AS400 con bases DB2, o para entornos .Net con bases de datos SQL Server. Se supone que el cliente móvil sería la parte innovadora del sistema, pero se puede desarrollar también dicho módulo para otras plataformas móviles.

- Como ya se ha comentado anteriormente, uno de los puntos débiles de la aplicación puede consistir en el consumo de batería del dispositivo. Una de las líneas más destacables a seguir podría ser un estudio del consumo de energía de estos dispositivos Android, dependiendo de la versión de la plataforma, modelo del dispositivo, tipo de proveedor que usen para obtener la ubicación del usuario, etc. y en general, de todo lo que pueda influir en el gasto de energía. De esta forma, podría realizarse un mejor ajuste de parámetros como por ejemplo, tiempo entre sincronizaciones con el servidor o el tiempo entre actualización de la localización del usuario. El objetivo sería minimizar el consumo de energía sin que se vieran afectadas las funcionalidades y objetivos principales de la herramienta.
- En todo momento, se ha destacado el carácter de integración de la herramienta con sistemas ya existentes. Por ello, se ha centrado la implementación en la gestión de las expediciones en reparto. Se podría implementar un segundo conjunto de módulos de utilidad para el resto de los procesos de la empresa, formando así un sistema más completo.

La aplicación ha sido probada en dos dispositivos reales, modelos HTC Magic con sistema operativo 2.2 y Motorola Milestone con versión 2.1 de la plataforma. Una tarea interesante sería probar la aplicaciones sobre otros dispositivos reales, con diversos tamaños de pantalla y diversas resoluciones.

Finalmente, se podrían realizar todo tipo de actualizaciones en la aplicación que representaran pequeñas mejoras de cada al usuario: nuevos diseños de interfaces gráficos más atractivos y configurables por el usuario, más opciones disponibles como por ejemplo mostrar en el mapa sólo las expediciones más cercanas a una determinada posición, adición de nuevos idiomas para seleccionar, etc.

Capítulo 6

PRESUPUESTO

Se presentan a continuación los cálculos del coste del proyecto. Para la realización de los mismos se ha tomado en cuenta los costes tanto de material como de mano de obra. Además, se añade también una sección sobre el coste por mantenimiento.

6.1. Coste de material

El material que se ha utilizado en el desarrollo de este proyecto ha sido:

- **Ordenador portátil Packard Bell Easynote TJ66**, con procesador Intel Core 2 Duo, procesador T6500 a 2,10 GHz, memoria RAM de 4GB y 320 GB de almacenamiento.
- **Dispositivo Motorola Milestone**, con un procesador OMAP3430 ARM Cortex A8 a una velocidad de 550 Mhz, memoria RAM de 512 Mb, pantalla TFT 3.7" WVGA (480 x 854 pixeles), GSM, GPRS, Wi-Fi y GPS, entre otras características.

En cuanto al *software*, se han utilizado los siguientes programas y librerías: IDE Eclipse, JDK Java 1.6, SDK Android, Plugin Android para Eclipse, librería kSoap2, base de datos mySQL, Apache Tomcat 6.0 y Windows Vista. De todo el *software* mencionado, únicamente el SO del ordenador portátil no es gratuito.

No se ha tenido en cuenta para este cálculo otros requisitos necesarios para el desarrollo, como por ejemplo el coste de las conexiones a internet necesarias tanto para el

portátil como para el terminal Android, por entenderse que estas son características que se poseerán independientemente de la realización del proyecto.

Concepto	**Precio (euros)**
Ordenador portátil	549
Dispositivo móvil	500
Windows Vista	155
Total	**1.204**

Cuadro 6.1: Coste del material.

El coste total de los materiales es de **1.204 euros**.

6.2. Coste de mano de obra

A continuación se incluyen los gastos que suponen los recursos humanos que intervienen en la gestión y desarrollo del proyecto, así como en la documentación de la memoria. Para ello, se estima necesario el trabajo de al menos dos personas: un ingeniero de telecomunicación y un ingeniero técnico de telecomunicación, a los cuales se les asigna unos salarios brutos mensuales de 2500 y 2000 euros respectivamente. Estimando el número de horas laborables al mes en un total de 160, resulta en un salario por hora de 15.625 euros para el ingeniero y de 12.5 euros para el ingeniero técnico.

En la siguiente tabla se muestra el coste total por persona y tarea desempeñada en el proyecto:

Tarea	**Trabajador**	**Horas**	**Coste (euros)**
Organización y gestión	Ingeniero	130	2.031,25
Documentación	Ingeniero	100	1.562,50
Estudio de las tecnologías	Ing. Técnico	80	1.000,00
Análisis y diseño	Ing. Técnico	30	375,00
Desarrollo y pruebas	Ing. Técnico	335	4187,50
Creación de la memoria	Ing. Técnico	80	1.000,00
Total			**10.156,25**

Cuadro 6.2: Coste de la mano de obra.

El coste total de personal asciende a **10.156,25 euros**.

6.3. Coste total

Una vez calculados los costes de los conceptos anteriores, se puede determinar el presupuesto para este proyecto, que se muestra en la siguiente tabla:

Costes	**Presupuesto (euros)**
Costes de material	1.204,00
Costes de mano de obra	10.156,25
Total	**11.360,25**

Cuadro 6.3: Coste total del proyecto.

En total el presupuesto de este proyecto se estima en **ONCE MIL TRESCIENTOS SESENTA EUROS CON VEINTICINCO CÉNTIMOS**.

6.4. Coste por servicio de mantenimiento

Adicionalmente, se presenta un importe por servicio de mantenimiento preventivo y correctivo. Éste coste incluiría defectos en el *software* desarrollado en este proyecto y no cubre, por tanto, evoluciones o cambios funcionales sobre el *software* desarrollado. Se abarcan todas las actividades necesarias para resolver incidencias, desde el diagnóstico de un problema detectado, hasta la resolución completa de todo el impacto producido en la aplicación.

En la siguiente tabla se muestra el coste del servicio de mantenimiento a nivel mensual y anual expresado en euros.

Servicios de mantenimiento	**Importe mensual**	**Importe mensual**
Disponibilidad del soporte (8x5)		
	500	**6.000**
Mantenimiento preventivo y correctivo (8x5)		

Cuadro 6.4: Coste del servicio de mantenimiento.

Apéndice A

Glosario de términos

- API: Application Programming Interface
- AVD: Android Virtual Device
- BD: Base de datos
- CDMA: Code Division Multiple Access
- DDMS: Dalvik Debug Monitor Service
- DVM: Dalvik Virtual Machine
- EDGE: Enhanced Data Rates for GSM Evolution
- ER: Entidad-Relación
- FDMA: Frecuency Division Multiple Access
- GPRS: General Packet Radio Service
- GPS: Global Positioning System
- GSM: Groupe Special Mobile
- ID: Identificador
- IDE: Integrated Development Environment
- J2ME: Java 2 Micro Edition
- JIT: Just In Time
- JSP: Java Server Pages

- LED: Light-Emitting Diode
- MIME: Multipurpose Internet Mail Extensions
- MMS: Multimedia Messaging Service
- OEM: Original Equipment Manufacturer
- PDC: Personal Digital Cellular
- PFC: Proyecto Fin de Carrera
- RAM: Random Access Memory
- REST: Representational State Transfer
- SDK: Software Development Kit
- SGBD: Sistema de gestión de base de datos
- SMS: Short Message Service
- SD: Secure Digital
- SO: Sistema Operativo
- SOAP: Simple Object Access Protocol
- SQL: Structured Query Language
- TDMA: Time Division Multiple Access
- UI: User Inteface
- UMTS: Universal Mobile Telecommunications System
- URI: Uniform Resource Identifier
- VM: Virtual Machine
- WAP: Wireless Application Protocol
- WAR: Web Archive
- WSDL: Web Services Description Language
- XML: EXtensible Markup Language

Apéndice B

Manual de usuario del cliente Android

Este apéndice pretende ser una guía para que el usuario aprenda el funcionamiento de la aplicación. En la figura 4.3 adjuntada en el capítulo de desarrollo del cliente Android se muestra un diagrama de navegación de la aplicación que permite formar una idea global del funcionamiento de la aplicación, la cual se complementará a continuación. Se explicará el interfaz gráfico de la herramienta comentando sus pantallas, menús y demás elementos gráficos.

Una vez instalada la aplicación en el terminal, el usuario podrá ejecutarla desde el menú de aplicaciones de Android. La aplicación puede ser usada tanto en orientación vertical como horizontal de la pantalla [*Fig. B.1*].

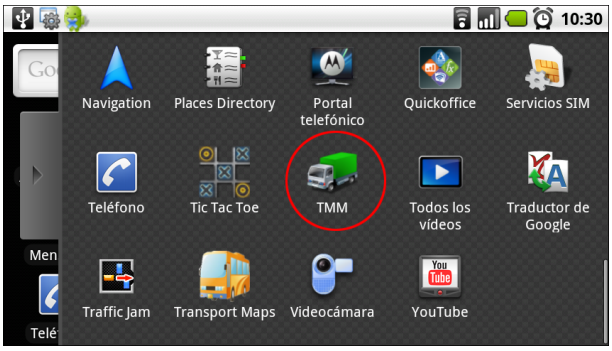

Figura B.1: [Manual Aplicación Android] Menú de aplicaciones del dispositivo.

PANTALLA DE INICIO DE SESIÓN

La primera pantalla de la herramienta es la de inicio de sesión [*Fig. B.2*]. Desde esta pantalla se puede cambiar el idioma seleccionando entre el inglés y el español, y también puede cerrarse la aplicación. Para comenzar a usar la herramienta se deberán rellenar los campos de usuario y contraseña previamente a pulsar el botón "Iniciar". En caso contrario se mostrará un mensaje indicándolo así.

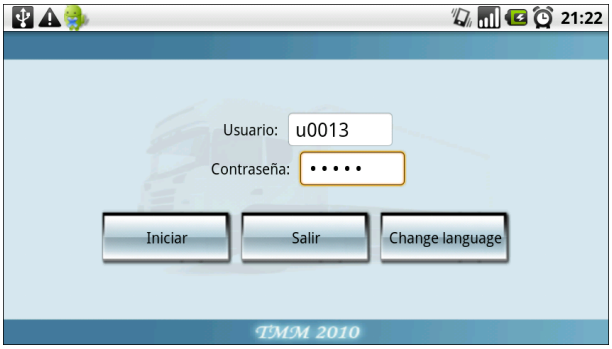

Figura B.2: [Manual Aplicación Android] Inicio de sesión.

Tras pulsar el botón de inicio de la aplicación, se visualizará un diálogo de espera en el que se le indica al usuario que sus datos están siendo sincronizados con el servidor [*Fig. B.3*]. En caso de que hubiera algún problema con la conexión, se informaría de ello con un mensaje por pantalla y volviendo a mostrar la pantalla de inicio.

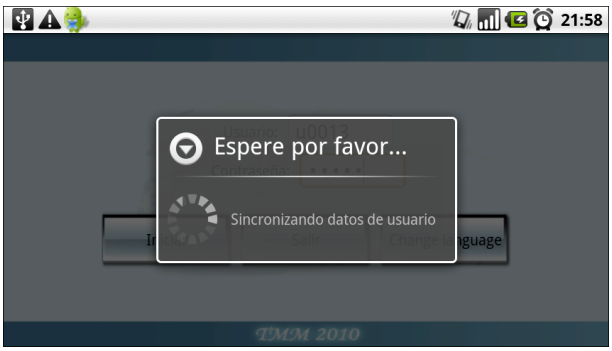

Figura B.3: [Manual Aplicación Android] Sincronización de datos.

MENÚ PRINCIPAL

Una vez que el terminal ha almacenado todos los datos relativos a la hoja de ruta del usuario, se muestra la pantalla del menú principal [*Fig. B.4*]. Esta pantalla muestra cuatro botones que dan acceso a los listados de expediciones, notificaciones y mapas. En el caso de las entregas, recogidas y notificaciones, se muestra además el número de cada una de ellas asignadas al usuario. Si éste pulsa la tecla de "Menú" del terminal, aparece un menú de opciones. Éstas permiten cambiar el idioma de la aplicación, forzar una sincronización en ese instante, cerrar la sesión y acceder a un menú expandido. El menú expandido permite abrir las aplicaciones de *Places Directory* y *Gasolineras España* [*Fig. B.5*].

Figura B.4: [Manual Aplicación Android] Menú principal.

En las figuras [*Fig. B.6*] y [*Fig. B.7*] se muestran las pantallas principales de las aplicaciones externas a las que el usuario tiene acceso directo desde la herramienta *TMM*.

Figura B.5: [Manual Aplicación Android] Menú expandido del menú principal.

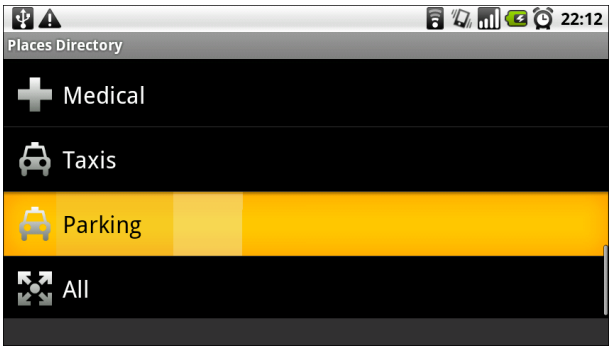

Figura B.6: [Manual Aplicación Android] Places Directory.

Figura B.7: [Manual Aplicación Android] Gasolineras España.

ENTREGAS

Desde el botón "Entregas" se accede al listado de entregas asociadas al usuario [*Fig. B.8*]. Cada entrega del listado va precedida por un icono circular con tres posibles colores dependiendo del estado de la entrega: verde si ha sido realizada, roja si no lo ha sido y gris si aún está pendiente. A la derecha del icono aparece el código de entrega y el cliente a quien va destinada.

Figura B.8: [Manual Aplicación Android] Listado de entregas.

Cuando se pulsa sobre una entrega de la lista, aparece un menú contextual con opciones variables dependiendo del estado de la entrega:

- **Entrega pendiente**: Se muestran las opciones "Más detalles", "Entrega realizada" y "Entrega no realizada".
- **Entrega realizada**: Las entregas realizadas se mostrarán en el listado hasta que sean sincronizadas con el servidor, momento en el que desaparecerán del mismo. En el transcurso de ese período, en el menú contextual aparece la única opción de "Más detalles".
- **Entrega no realizada**: Una entrega con este estado permanecerá visible aunque se produzcan sincronizaciones. Estas entregas tienen la posibilidad de poder realizarse con posterioridad y por ello muestra las opciones de "Más detalles" y "Entrega realizada".

La pantalla de detalle de entrega muestra toda la información relativa a una entrega [*Fig. B.9*].

Figura B.9: [Manual Aplicación Android] Detalle de una entrega.

La pantalla de reporte de entrega realizada recoge si ha habido alguna incidencia en la entrega, posibles comentarios del transportista y otros datos como fecha y hora que no son modificables [*Fig. B.10*]. Además, avisa al transportista si la entrega tiene portes o reembolso asociados [*Fig. B.11*]. Las posibles incidencias a seleccionar son "Daños" y "Faltas/Sobras".

La pantalla de entrega no realizada permite introducir comentarios al usuario para que sean enviados junto con datos de la fecha y hora, los cuales no son modificables [*Fig. B.12*].

RECOGIDAS

La funcionalidad de las recogidas es muy similar a la de las entregas pero con algunas variaciones. Desde el menú principal se accede al listado de recogidas [*Fig. B.13*]. Cada recogida en el listado está precedida por un icono circular con cuatro posibles colores seguún si la recogida está pendiente, realizada, no realizada o rechazada. Además de mostrar el código de recogida y cliente, se muestra una segunda línea en la que se visualiza el horario.

Según el estado de la recogida, se le asocian distintas opciones en el menú contextual que aparece al seleccionarla:

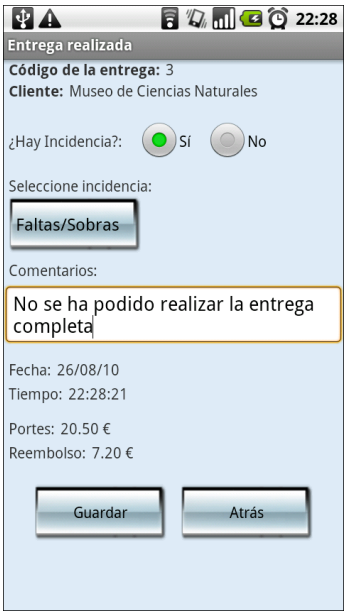

Figura B.10: [Manual Aplicación Android] Entrega realizada.

Figura B.11: [Manual Aplicación Android] Aviso de entrega con reembolso.

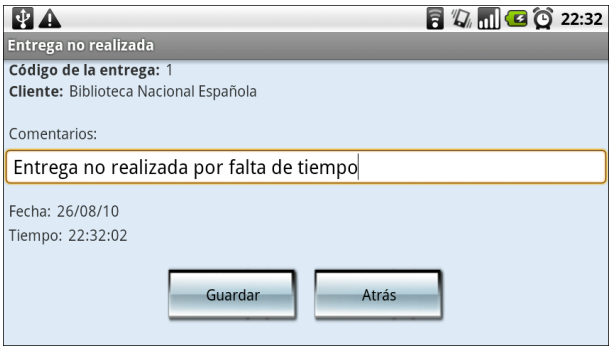

Figura B.12: [Manual Aplicación Android] Entrega no realizada.

Figura B.13: [Manual Aplicación Android] Listado de recogidas.

- **Recogida pendiente**: Se le asocian todas las opciones disponibles, que son "Más detalles", "Recogida realizada", "Recogida no realizada" y "Recogida rechazada".
- **Recogida realizada**: El usuario sólo podrá consultar los detalles de una entrega realizada. La opción para ello es "Más detalles"
- **Recogida no realizada**: Al igual que para las entregas, una recogida no realizada puede ser realizada posteriormente (aunque no rechazada). Las opciones en este caso serán "Más detalles", "Recogida realizada".
- **Recogida rechazada**: Una entrega que ha sido rechazada tiene asociada la opción única de consultar sus detalles.

Las recogidas que han sido realizadas desaparecerán del listado cuando sean sincronizadas con el servidor. El resto de las recogidas seguirán persistiendo en el dispositivo aunque se realicen sincronizaciones.

La pantalla de detalle de una recogida es similar a la de una entrega, exceptuando algunas variaciones en los campos visibles.

En cuanto a la pantalla de "recogida realizada", el usuario puede indicar en número de bultos del que se compone la recogida, así como introducir comentarios personales sobre la misma [*Fig. B.14*]. Datos como la fecha y la hora no serán modificables por el usuario.

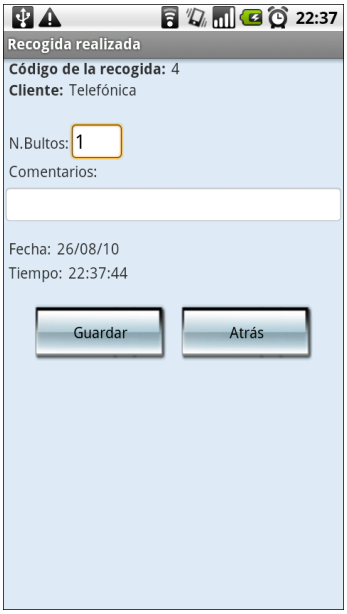

Figura B.14: [Manual Aplicación Android] Recogida realizada.

En las pantallas de "recogida no realizada" y "recogida rechazada" el usuario seleccionará el motivo por el que la recogida no se ha realizado o se ha rechazado. Los motivos disponibles para seleccionar en caso de una recogida no realizada son "Cerrado", "No preparado", "No tienen nada" y "Recogida especial". En el caso de una recogida rechazada [*Fig. B.15*], se podrán seleccionar los motivos "Falta de espacio", "Falta de tiempo" y "Error fuera de ruta". El usuario podrá introducir comentarios personales en el reporte en ambos casos, que se enviarán junto con la fecha y la hora.

NOTIFICACIONES

Al igual que para las expediciones, al listado de notificaciones se accede desde un botón del menú principal. En este listado se muestra en cada línea el texto de una notificación, de forma que si es demasiado largo se termina la línea con puntos suspensivos.

Las notificaciones sólo pueden ser leídas y borradas. Las que aún no han sido leídas aparecen en negrita y las leídas en texto normal. Las notificaciones que han sido borradas aparecen tachadas hasta que se sincronicen con el servidor, momento en que desaparecerán del listado. Además, se ordenan de forma que las notificaciones no leídas aparecen en la parte superior de la lista, seguidas de las leídas y de las borradas [*Fig. B.16*].

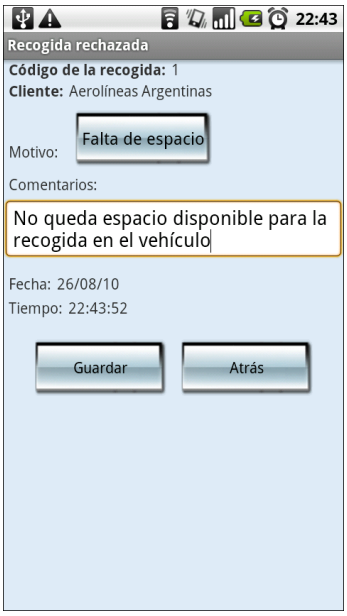

Figura B.15: [Manual Aplicación Android] Recogida rechazada.

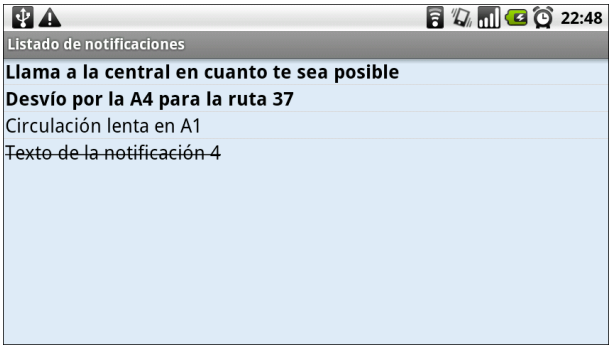

Figura B.16: [Manual Aplicación Android] Listado de notificaciones.

Cuando el usuario seleciona una de las notificaciones de la lista, tiene la opción de leerla o borrarla. En caso de pulsar la opción de "Borrar", se le pedirá confirmación sobre la operación. Si en cambio pulsa "Leer", aparecerá una pantalla con el texto completo de la notificación [*Fig. B.17*], en la cual se le vuelve a ofrecer la opción de borrar la notificación.

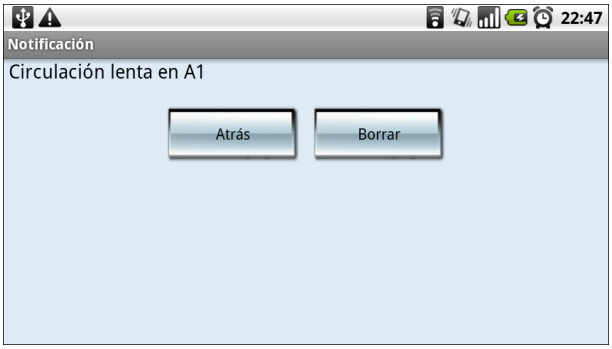

Figura B.17: [Manual Aplicación Android] Pantalla de notificación.

MAPA

El botón "Mapa" de la pantalla del menú principal abre una pantalla que muestra un mapa en el que se pueden visualizar todas las expediciones asignadas al usuario situadas en sus coordenadas geográficas [*Fig. B.18*]. Además, dichas expediciones son representadas por letras y colores para diferencias el tipo y el estado de las mismas. Junto a las entregas y notificaciones, se muestra también la ubicación del usuario mediante un icono parpadeante.

La pantalla tiene un menú con las siguientes opciones:

- **Mi ubicación**: Al pulsar esta opción, el mapa se centra en la ubicación del usuario y muestra un icono que lo representa.
- **Opciones**: Abre la pantalla de opciones del mapa.
- **Ver todo**: Muestra en el mapa tanto las entregas como recogidas.
- **Sólo entregas**: Se muestran sólo entregas.
- **Sólo recogidas**: Se muestran solo recogidas.

En la pantalla de opciones del mapa hay tres parámetros configurables:

Figura B.18: [Manual Aplicación Android] Menú de opciones del mapa.

- Vista del mapa: Permite elegir entre una vista mapa, vista satélite y la *street view*.
- Expediciones a mostrar: El usuario puede elegir en un menú contextual entre mostrar todas las expediciones independientemente de su estado o que se muestren solo aquellas que se encuentren en un determinado estado.
- Modo de pantalla: La pantalla del mapa puede mostrarse con barra de estado o sin ella.

Es importante destacar que las opciones de "Solo entregas" y "Solo recogidas" del menú de opciones de la pantalla no son independientes de la selección elegida sobre las expediciones en la pantalla de opciones. Es decir, si en la pantalla de opciones se ha elegido mostrar sólo las expediciones no realizadas, cuando el usuario pulse "Solo entregas" lo que se mostrará en el mapa serán las entregas cuyo estado sea "no realizada" y no todo el conjunto de recogidas.

El usuario puede manejar el mapa en todo momento haciendo *zoom* para acercarlo o alejarlo, moverlo hacia la zona en la que esté interesado, mostrando sólo ciertas expediciones, etc. Cuando el usuario pulsa sobre una de las expediciones en el mapa, aparece un diálogo informativo mostrando el código de la expedición, el cliente, dirección y estado de la misma. Esto es así cuando la vista de mapa seleccionada es "vista mapa" o "vista satélite", pero si la vista seleccionada es la vista "street view", la funcionalidad del diálogo se amplia. Además de la información anterior, también aparece un botón para

visualizar la dirección de la expedición con las vistas del servicio "street view" de Google, pudiendo ver una imagen real del punto geográfico concreto.

En la figura [*Fig. B.19*] se muestra el diálogo informativo en la vista "street view" y en la figura [*Fig. B.20*] se muestra la vista "street view" de la dirección.

Figura B.19: [Manual Aplicación Android] Información de expedición en el mapa.

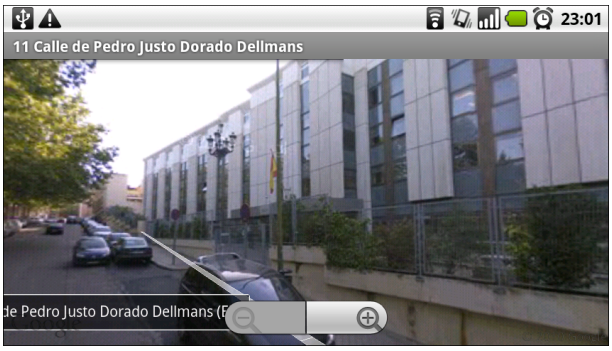

Figura B.20: [Manual Aplicación Android] *Street View*.

Apéndice C

Manual de usuario del perfil de gestión

INICIO DE SESIÓN

El usuario podrá acceder a todas las funcionalidades de la herramienta a través de la pantalla de inicio de sesión, en la que deberá introducir su código de usuario y contraseña [*Fig. C.1*]. Solo los perfiles "Administrador" serán usuarios válidos para este módulo.

Figura C.1: [Manual del perfil de gestión] Inicio de sesión.

MENÚ PRINCIPAL

Tras hacer *login*, se mostrará la pantalla del menú principal [*Fig. C.2*] con los siguientes elementos:

- **Consultar entregas**: Muestra el listado de entregas.
- **Consultar recogidas**: Muestra el listado de recogidas.
- **Gestionar usuarios**: Muestra el listado de usuarios.
- **Gestionar notificaciones**: Muestra el listado de notificaciones.
- **Cerrar sesión**: Cierra la sesión en la aplicación.

En la esquina superior derecha de la pantalla aparecerá en todas las pantallas de la herramienta el código del usuario cuya sesión está abierta.

Figura C.2: [Manual del perfil de gestión] Menú principal.

CONSULTAR ENTREGAS

La pantalla del listado de entregas muestra una tabla en la que aparecerán todas las entregas registradas en el sistema [*Fig. C.3*]. Para cada una de ellas se muestran los códigos de entrega, usuario y oficina, el estado, la fecha, hora y observaciones del reporte y, por último, las posibles incidencias asociadas. Toda esta información en cada fila de la tabla está acompañada de un código de color asociado al estado de la entrega, de forma que resulte rápido reconocer dicho estado puesto que esta información es la más relevante. Dicho código de color se muestra en todo momento mediante un icono circular al inicio de la fila y en el color de fondo de la misma en el momento en que el cursor pasa por encima de dicha fila. Según el estado, los colores asociados son:

- Entrega pendiente: Color blanco.
- Entrega realizada: Color verde.
- Entrega no realizada: Color rojo.

Figura C.3: [Manual del perfil de gestión] Listado de entregas.

Para facilitar la tarea de consulta de una o varias entregas en concreto, el usuario dispone de un filtro de búsqueda situado sobre la tabla de entregas. Los campos por los que el usuario puede filtrar la información son los códigos de entrega, usuario y oficina, el estado, la fecha de reporte y la incidencia. Los campos se pueden usar de forma conjunta, es decir, que si se usa el campo código de oficina y estado, se mostrarán todas las entregas perteneciente a esa oficina cuyo estado sea el indicado en el filtro.

Cuando se selecciona una entrega pulsando sobre ella, se navega a la pantalla del detalle de la entrega [*Fig. C.4*]. En dicha pantalla se muestra una tabla con toda la información de la entrega, de forma que los campos que no se mostraban en la tabla del listado son accesibles desde esta nueva tabla.

Figura C.4: [Manual del perfil de gestión] Detalle de una entrega.

Pulsando el botón “Atrás” situado en la parte inferior de la tabla se regresa al listado de entregas.

CONSULTAR RECOGIDAS

La pantalla del listado de recogidas muestra una tabla en la que aparecerán todas las recogidas registradas en el sistema [*Fig. C.5*]. Para cada una de ellas se muestran los códigos de recogida, usuario y oficina, el estado, la fecha, hora y observaciones del reporte y, por último, las posibles incidencias asociadas. Toda esta información en cada fila de la tabla está acompañada de un código de color asociado al estado de la recogida, de forma que resulte rápido reconocer dicho estado puesto que esta información es la más relevante. Dicho código de color se muestra en todo momento mediante un icono circular al inicio de la fila y en el color de fondo de la fila en el momento en que el cursor pasa por encima de dicha fila. Según el estado, los colores asociados son:

- Recogida pendiente: Color blanco.
- Recogida realizada: Color verde.
- Recogida no realizada: Color rojo.
- Recogida rechazada: Color amarillo.

Para facilitar la tarea de consulta de una o varias recogidas en concreto, el usuario dispone de un filtro de búsqueda situado sobre la tabla de recogidas. Los campos por los que el usuario puede filtrar la información son los códigos de recogida, usuario y oficina, el estado, la fecha de reporte y la incidencia. Los campos se pueden usar de forma conjunta, es decir, que si se usa el campo código de usuario y estado, se mostrarán todas las entregas asociadas a ese usuario cuyo estado sea el indicado en el filtro.

En la parte inferior de la tabla se muestra un botón “Atrás” mediante el cual se regresa al menú principal de la herramienta.

Cuando se selecciona una recogida pulsando sobre ella, se navega a la pantalla del detalle de la recogida, que es similar a la del detalle de entregas pero con los campos propios de una recogida. Todos los campos que no se mostraban en la tabla del listado son accesibles desde esta nueva tabla. Mediante el botón “Atrás” situado en la parte inferior de la tabla se navega de regreso al listado de recogidas.

Figura C.5: [Manual del perfil de gestión] Listado de recogidas.

GESTIONAR USUARIOS

En la pantalla de gestión de usuarios se muestra una tabla con los campos de un usuario: códigos de oficina y usuario, contraseña, nombre y apellidos, el perfil y los minutos de sincronización asociados a cada uno de ellos [*Fig. C.6*]. Para facilitar la labor de búsqueda de un determinado usuario, se pueden filtrar las filas de la tabla mediante el filtro que se muestra en la parte superior de la misma.

Desde esta pantalla se pueden crear, modificar y eliminar usuarios. Además, también se accede a mapas asociados a cada usuario en los que se muestra información sobre su actividad.

Mediante el botón "Atrás" situado en la parte inferior de la tabla se vuelve al menú principal de la aplicación.

Figura C.6: [Manual del perfil de gestión] Listado de usuarios.

Pulsando sobre un usuario del listado se navega a la pantalla del mapa asociado a dicho usuario. Este mapa no es estático, es decir, el usuario podrá arrastrarlo en cualquier dirección, hacer *zoom* para acercarlo o alejarlo y cambiar las vistas del mapa entre modo mapa, modo satélite o modo híbrido.

Sobre el mapa se muestran gráficamente las expediciones que el transportista en cuestión ha reportado. Éstas son representadas mediante iconos de colores para diferenciarlas por tipo y estado. El usuario puede pulsar sobre cualquiera de las expediciones para visualizar información sobre las mismas, en concreto, el código de expedición, estado, cliente, dirección y fecha y hora del reporte.

Los dispositivos de los transportistas almacenan y envían al servidor la posición geográfica de los mismos cada cierto período de tiempo. La utilidad de estas coordenadas geográficas es mostrar las rutas que han ido haciendo los transportistas en su actividad. Estas rutas son representadas en los mapas mediante líneas azules.

Figura C.7: [Manual del perfil de gestión] Mapa.

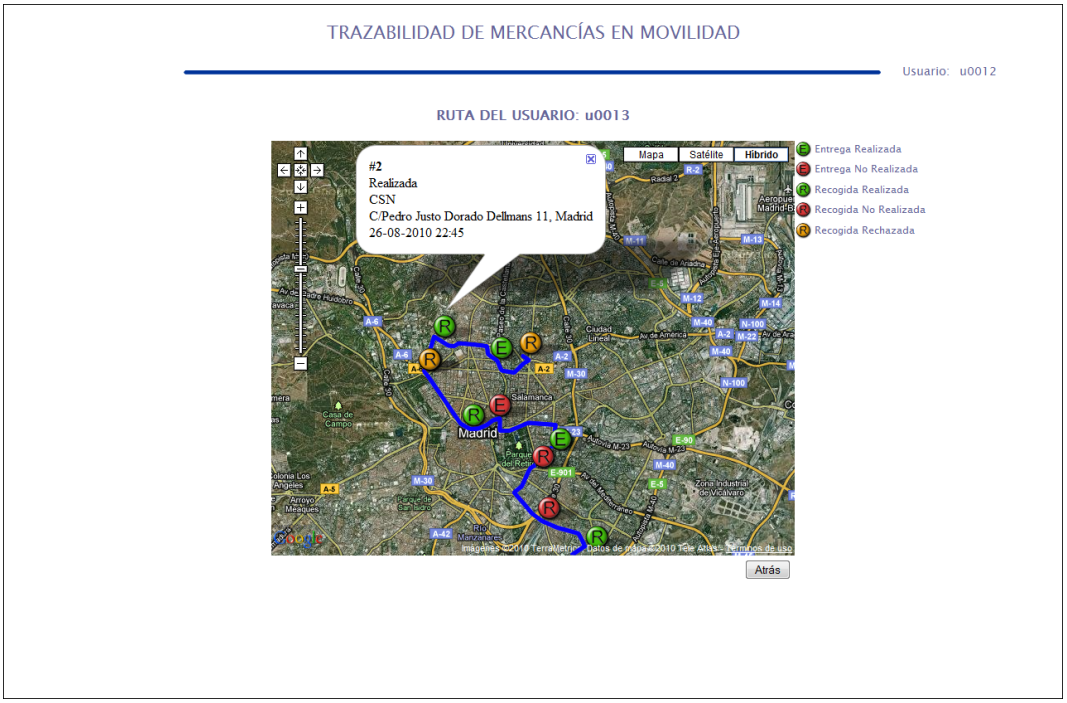

Figura C.8: [Manual del perfil de gestión] Mapa en vista híbrida.

Desde el botón "Nuevo" situado en la parte inferior izquierda de la tabla se accede al formulario de creación de un nuevo usuario. Esta pantalla y la de modificación de usuario son similares, pero en el segundo caso los campos aparecen completados con los datos del usuario que se modifica, siendo editables todos menos el código de dicho usuario. En el alta de un usuario todos los campos son editables.

Figura C.9: [Manual del perfil de gestión] Modificar un usuario.

Una vez completados los campos con la información deseada, se pulsará el botón "Guardar" para almacenar los cambios hechos para un usuario que se estaba modificando o para almacenar un nuevo usuario en el caso de no existir previamente. Automáticamente se regresa al listado de usuarios tras pulsar dicho botón, pudiendo visualizar los cambios hechos. También es posible cancelar la operación pulsando el botón "Cancelar", tras lo cual se regresa al listado.

Para acceder a la pantalla de modificación de un usuario concreto es necesario pulsar sobre el icono del lápiz que aparece en su fila correspondiente. Seguidamente a la derecha de éste se muestra un segundo icono. En este caso, la función que realiza este segundo icono es la eliminación de un usuario. Previamente al borrado aparece un diálogo de confirmación para dicha operación. Para una respuesta afirmativa a la confirmación, el usuario es eliminado de la base de datos, el diálogo desaparece y se refresca el listado de usuarios con la información actualizada, es decir, sin la fila correspondiente al usuario eliminado.

GESTIONAR NOTIFICACIONES

El listado de notificaciones es una tabla con todas las notificaciones creadas para los usuarios [*Fig. C.10*]. En ella se muestran los campos de los códigos de notificación, usuario y oficina, estado, texto del mensaje y por último, la fecha y hora del reporte. Es posible filtrar filas de la tabla por los campos de código de la oficina y código de usuario.

El estado que puede tener una notificación es "No recibida", "No leída", "Leída" y "Borrada".

Figura C.10: [Manual del perfil de gestión] Modificar un usuario.

Aunque en esta tabla se muestra el campo "Texto" del mensaje, sólo aparecen las primeras palabras del mismo para no producir filas demasiado grandes. Al pulsar sobre una de las notificaciones, aparece una ventana emergente en la que se muestra el texto al completo de la notificación [*Fig. C.11*].

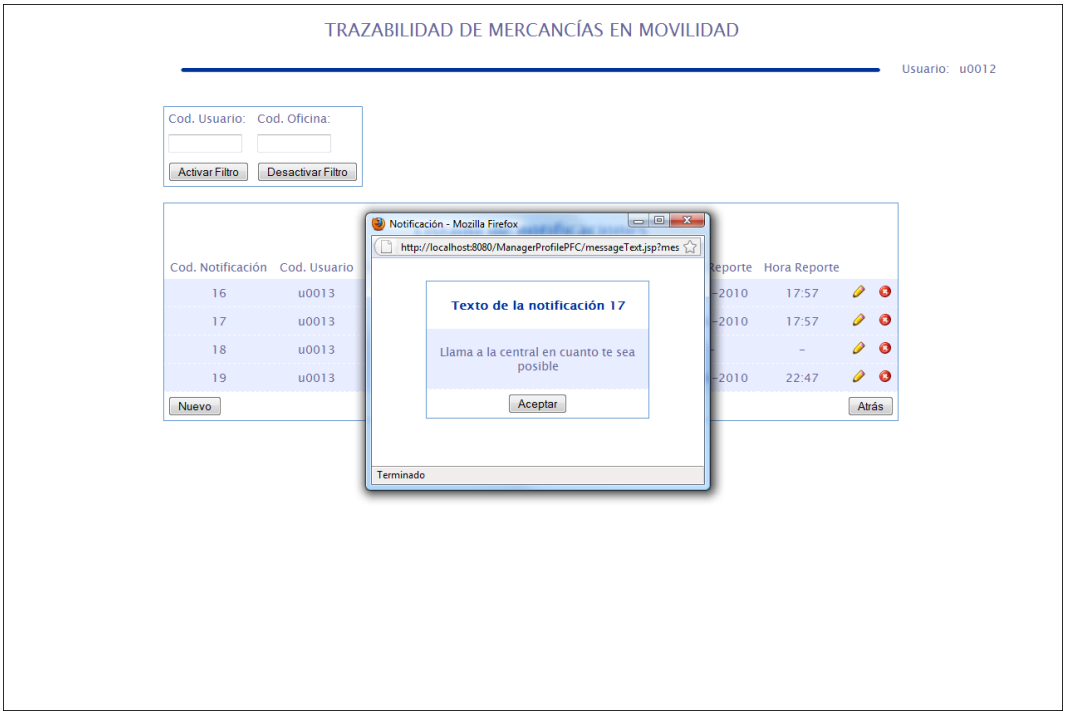

Figura C.11: [Manual del perfil de gestión] Texto de una notificación.

Al igual que en el caso del listado de usuarios, desde esta pantalla también se pueden crear, modificar y borrar notificaciones. Para su creación es necesario pulsar el botón "Nueva" de la parte inferior izquierda de la tabla. La pantalla para crear una notificación es muy simple, se compone de un campo para introducir el texto del mensaje (no mayor a ciento sesenta caracteres), el usuario destinatario del mensaje y la oficina asociada al mismo [*Fig. C.12*]. Tras guardar la nueva notificación, se regresa automáticamente al listado de notificaciones donde se visualizará la nueva notificación creada.

A la pantalla de modificación de una notificación se accede pulsando el icono del lápiz de la fila asociada a la notificación que se desea modificar. Esta pantalla es como la explicada anteriormente pero los campos aparecen completados con la información de la notificación seleccionada.

Para borrar del sistema una notificación se debe pulsar el icono circular rojo que aparece a la derecha del icono del lápiz. Se mostrará al usuario un diálogo de confirmación de borrado [*Fig. C.13*], tras lo que se borrará la notificación de la base de datos y se actualizará el listado de notificaciones de forma que desaparecerá la fila de la notificación eliminada.

Figura C.12: [Manual del perfil de gestión] Creación de una notificación.

Figura C.13: [Manual del perfil de gestión] Borrado de una notificación.

Apéndice D

Manual del perfil de introducción de datos

INICIO DE SESIÓN

Este módulo de la herramienta permitirá introducir nuevas entregas y recogidas en el sistema. Para ello, el usuario deberá iniciar sesión introduciendo su código de usuario y contraseña para que sean validados. Los únicos usuarios con acceso a esta herramienta serán los que tengan asignado el perfil "DataEntry".

La siguiente pantalla en mostrarse es una pantalla de menú, donde se muestran los elementos "Nueva entrega", "Nueva recogida" y "Cerrar sesión". En la esquina superior derecha de la pantalla se muestra en todas las pantallas de la herramienta el código de usuario que ha sido validado en la pantalla de *login*.

Las pantallas de crear nuevas entregas y recogidas son similares. En ambos casos se presenta al usuario un formulario con todos los campos asociados a cada tipo de expedición. Mediante los botones "Guardar" y "Cancelar" se puede guardar la nueva expedición o cancelar la operación. En cualquiera de los casos se volverá automáticamente al menú principal.

Figura D.1: [Manual del perfil de introducción de expediciones] Inicio de sesión.

Figura D.2: [Manual del perfil de introducción de expediciones] Menú principal.

Figura D.3: [Manual del perfil de introducción de expediciones] Nueva entrega.

Figura D.4: [Manual del perfil de introducción de expediciones] Nueva recogida.

Bibliografía

[1] http://www.cbsnews.com/stories/2010/05/21/60minutes/main6506912.shtml?tag=contentMain;contentBody

[2] http://www.cabinas.net/monografias/tecnologia/generaciones_de_la_telefonia_celular.asp

[3] http://www.mailxmail.com/curso-telefonia-celular-movil-funcionamiento-generaciones/telefonia-celular-tercera-generacion-3g

[4] http://es.wikipedia.org/wiki/Telefonía_móvil_3G

[5] http://www.theinquirer.es/2010/04/30/motorola-renace-gracias-a-android.html

[6] http://www.itespresso.es/motorola-sorprende-con-beneficios-de-69-millones-de-dolares-44848.html

[7] http://metrics.admob.com/wp-content/uploads/2009/05/admob-mobile-metrics-april-09.pdf

[8] http://metrics.admob.com/wp-content/uploads/2010/05/AdMob-Mobile-Metrics-Apr-10.pdf

[9] http://www.google.com.sv/corporate/execs.html#jeff

[10] http://investor.google.com/webcast.html

[11] http://www.apache.org/licenses/LICENSE-2.0.html

[12] http://www.openhandsetalliance.com/oha_overview.html

[13] http://open.movilforum.com/blog/introduccion-android

[14] http://android-developers.blogspot.com/2010/05/dalvik-jit.html

[15] http://www.youtube.com/watch?v=ptjedOZEXPM

[16] http://www.packetvideo.com/products/android/index.html

[17] http://developer.android.com/guide/basics/what-is-android.html

[18] http://developer.android.com/guide/topics/fundamentals.html

[19] http://developer.android.com/guide/topics/intents/intents-filters.html

[20] http://developer.android.com/guide/topics/resources/providing-resources.html

[21] http://developer.android.com/guide/topics/manifest/manifest-intro.html

[22] http://developer.android.com/guide/topics/security/security.html

[23] http://code.google.com/intl/es-ES/android/add-ons/google-apis/mapkey.html

[24] http://developer.android.com/guide/topics/location/index.html

[25] http://developer.android.com/guide/developing/tools/avd.html

[26] http://developer.android.com/guide/developing/tools/ddms.html

[27] http://www.softwarelibre.gob.ve/wiki-sl/index.php/Servidor_Tomcat

[28] http://www.programacion.com/articulo/tomcat__introduccion_134#1_ficherosconfiguracion

[29] http://www.eclipsetotale.com/tomcatPlugin.html

[30] http://www.jorgesanchez.net/bd/bdrelacional.pdf

[31] http://www.uaem.mx/posgrado/mcruz/cursos/miic/MySQL.pdf

[32] http://warp.es/mysql/productos/razones/

[33] http://www.scribd.com/doc/3321228/JDBC

[34] http://pedrorojas.over-blog.es/pages/Conectividad_JDBC-1356910.html

[35] http://www.mobialia.com/

[36] http://code.google.com/intl/es/apis/maps/terms.html#section_10_9

[37] http://petra.euitio.uniovi.es/ i6950404/wiki/pmwiki.php?
n=Tema6.RESTVSSOAP

[38] http://ksoap2.sourceforge.net/

[39] http://developer.android.com/resources/dashboard/platform-versions.html

[40] http://code.google.com/intl/es/apis/maps/documentation/utilities/
polylinealgorithm.html

[41] http://facstaff.unca.edu/mcmcclur/GoogleMaps/EncodePolyline/

[42] http://es.wikipedia.org/wiki/Android

[43] http://www.instamapper.com/overview.html

[44] http://revista.eia.edu.co/articulos12/EIA %2012 %20 %28pag. %2023-38 %29.pdf